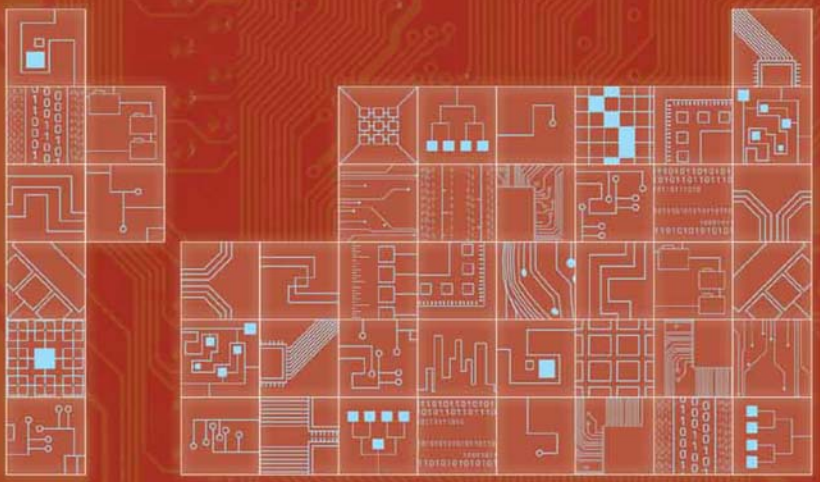
Edited by
Brian Bailey, Grant Martin and Thomas Anderson

Taxonomies
for the Development and Verification of
Digital Systems



Springer

VISIT…

Taxonomies for the Development and Verification of Digital Systems

Edited by
Brian Bailey
Grant Martin
Thomas Anderson

Taxonomies for the Development and Verification of Digital Systems

Springer

Library of Congress Cataloging-in-Publication Data

A C.I.P. Catalogue record for this book is available
from the Library of Congress.

ISBN 0-387-24019-5 e-ISBN 0-387-24021-7 Printed on acid-free paper.

© 2005 Springer Science+Business Media, Inc.
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, Inc., 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now know or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks and similar terms, even if the are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed in the United States of America.

9 8 7 6 5 4 3 2 1 SPIN 11329473

springeronline.com

Table of Contents

Abbreviations and Acronyms xi
Contributors xv
Preface xix
Chapter 1 Introduction 1
1.1 What is a Taxonomy? 1
1.2 About this Collection of Taxonomies 1
1.3 The History of the Taxonomies 2
1.3.1 Early Model Taxonomy Work 2
1.3.2 First Model Taxonomy Industrial Effort 3
1.3.3 Work within the VSIA 4
1.3.4 Extension into other Areas 4
1.3.5 Latest Revisions 5
1.4 Taxonomy Organization 5
1.4.1 Use of this Book 6
Chapter 2 Model Taxonomy 9
2.1 Introduction 9
2.1.1 Taxonomy Definition 9
2.1.1.1 Temporal Resolution Axis 12
2.1.1.2 Data Resolution Axis 13
2.1.1.3 Functional Resolution Axis 14
2.1.1.4 Structural Resolution Axis 15
2.1.1.5 Software Programming Resolution Axis 16
2.1.2 Internal/External (Interface) Concept 17
2.1.3 Note on Structure/Behavior/Interface Concepts 19
2.1.4 Additional Attributes 21
2.1.5 Vocabulary 21
2.2 General Modeling Concepts 22
2.2.1 Primary Model Classes 23
2.2.1.1 Functional Model 24
2.2.1.2 Behavioral Model 25
2.2.1.3 Structural Model 26
2.2.1.4 Interface Model 27
2.2.1.5 System-Level Interfaces 28
2.2.2 Specialized Model Classes 30
2.2.2.1 Performance Model 30
2.2.2.2 Mixed-Level Model 31

2.2.3 Computation Model Classes 32
2.2.3.1 Dataflow Graph Model 32
2.2.3.2 Other Computational Models 33
2.3 Other System Models 35
2.3.1 Executable Specification 35
2.3.2 Mathematical-Equation Model 36
2.3.3 Algorithm Model 37
2.4 Architecture Models 38
2.4.1 Token-Based Performance Model 38
2.4.2 Abstract-Behavioral Model 39
2.4.3 Dataflow Graph (DFG) Task Primitive 41
2.4.3.1 Instruction-Set Architecture (ISA) Model 41
2.5 Hardware Models 42
2.5.1 Detailed Behavioral Model 42
2.5.2 Register-Transfer-Level (RTL) Model 43
2.5.3 Logic-Level Model 44
2.5.4 Cell-Level Model 45
2.6 Switch-Level Model 46
2.6.1 Circuit-Level Model 46
2.7 Implementation-Level Performance Models 47
2.7.1 Basic Delay Model 47
2.7.2 Timing Analysis Model 48
2.7.3 Power Model 48
2.7.4 Peripheral-Interconnect Model 49
2.8 Software Models 49
2.8.1 Requirements Modeling 50
2.8.2 Pseudo-Code 50
2.8.3 High-Level Language (HLL) 50
2.8.4 Assembly Code 51
2.8.5 Microcode 51
2.8.6 Object Code 52
2.9 Supporting Terms 52
2.9.1 Abstraction Level and Hierarchy 52
2.9.1.1 Abstraction Level 52
2.9.1.2 Hierarchy 53
2.9.2 Design Object Classes 54
2.9.2.1 System 54
2.9.2.2 Component, Module 55
2.9.2.3 Architecture 56
2.9.2.4 Structure 57
2.9.2.5 Hardware 57
2.9.2.6 Software 57

2.9.2.7 Firmware....57
2.9.3 Information Classes....58
2.9.3.1 Application Data....58
2.9.3.2 Control Data....58
2.9.4 Design Process Terms....58
2.9.4.1 Synthesis....58
2.9.4.2 Simulation....58
2.9.4.3 Emulation....59
2.9.4.4 Interface-Based Design....59
2.9.4.5 Top-Down Design....59
2.9.4.6 Prototype....60
2.9.4.7 Physical Prototype....60
2.9.4.8 Virtual Prototype....61
2.9.4.9 Virtual Prototyping....61
2.9.5 Design-Tool Terms....62
2.9.5.1 Model....62
2.9.5.2 Emulator....62
2.9.5.3 Simulator....62
2.9.6 Verification and Test-Related Terms....62
2.9.6.1 Testbench....62
2.9.6.2 Test Vector....62
2.9.6.3 Functional Test....63
2.9.6.4 Operational Test....63
2.9.6.5 Boundary Scan....63
2.9.6.6 Signature Analysis....63
2.9.7 Requirements and Specifications....63
2.9.7.1 Specification....63
2.9.7.2 Executable Specification (E-Spec)....64
2.9.7.3 Requirement Specification (Req-Spec)....64
2.9.7.4 Executable-Requirement Specification (ER-Spec)....64
2.9.7.5 Design Specification (Design-Spec)....64
2.9.7.6 Executable Design Specification (ED-Spec)....64
2.9.8 Reusability and Interoperability....65
2.9.8.1 Reusability....65
2.9.8.2 Model Interoperability....65
2.9.9 Interface-Related Terms....65
2.9.9.1 General Interface Terms....65
Chapter 3 Functional Verification Taxonomy....69
3.1 Introduction....69
3.1.1 Classifications of Verification....69
3.1.2 Definitions....70
3.2 Intent Verification....71

3.2.1 Dynamic Verification 71
3.2.1.1 Deterministic Simulation 71
3.2.1.2 Random Pattern Simulation 72
3.2.1.3 Hardware Acceleration 72
3.2.1.4 Hardware Modeling 73
3.2.1.5 Monitors 73
3.2.1.6 Protocol Checkers 73
3.2.1.7 Expected Results Checkers 73
3.2.2 Static Functional Verification 74
3.2.3 Formal Verification 74
3.2.3.1 Property/Model Checking 74
3.2.3.2 Theorem Proving 75
3.2.4 Dynamic-Formal Hybrid Verification 76
3.2.4.1 Symbolic Simulation 76
3.2.4.2 Dynamic Formal Verification 76
3.2.4.3 Formal Coverage 77
3.2.4.4 Formal Constraint-Driven Stimulus Generation 77
3.2.5 Hardware/Software Co-Verification 77
3.2.6 Emulation 78
3.2.7 Physical Prototyping 79
3.2.7.1 Emulation Systems 79
3.2.7.2 Reconfigurable Prototyping System 80
3.2.7.3 Application-Specific Prototype 80
3.2.8 Virtual Prototyping 80
3.2.9 Verification Metrics 81
3.2.9.1 Hardware Code Coverage 81
3.2.9.2 Functional Coverage 82
3.2.10 Definitions 82
3.3 Equivalence Verification 87
3.3.1 Dynamic Verification 87
3.3.1.1 Deterministic Simulation 87
3.3.1.2 Expected Results Checkers 87
3.3.1.3 Golden Model Checkers 88
3.3.1.4 Regression Testing 88
3.3.1.5 Verification Test Suite Migration 88
3.3.2 Formal Equivalence Checking 90
3.3.2.1 Boolean Equivalence Checking 90
3.3.2.2 Sequential Equivalence Checking 90
3.3.3 Physical Verification 91
3.3.4 Definitions 92
3.4 VC Verification 93
3.5 Integration Verification 94

3.6 Functional Verification Mapping 94
3.7 Summary 96
Chapter 4 Platform-Based Design 103
4.1 Platform-Based Design 103
4.1.1 Introduction 105
4.1.2 Background and History 105
4.1.3 Platform-Based Development System 106
4.2 Platform Taxonomies 108
4.2.1 Platform Object Complexity 108
4.2.1.1 Complexity Levels 108
4.2.1.2 Interfaces 111
4.2.2 Platform Specification Approaches 113
4.2.2.1 Technology-Driven (Bottom-Up) Specification 114
4.2.2.2 Architecture-Driven (Middle-Out) Specification 115
4.2.2.3 Application-Driven (Top-Down) Specification 116
4.2.2.4 Platform Specification Attributes 116
4.2.2.5 Metrics for Platform Specification Approaches 125
4.2.2.6 Alignment with Platform Specification Approaches 127
4.2.2.7 On The Evolution of Platform Specifications 129
4.3 Definitions 131
Chapter 5 Hardware-dependent Software 135
5.1 Introduction 135
5.1.1 Purpose of this Chapter 135
5.1.2 Intended Audience 136
5.2 HdS Terms and Abbreviations 136
5.2.1 Basic HdS Definitions 137
5.2.1.1 HdS (Hardware-dependent Software) 137
5.2.1.2 HAL (Hardware Abstraction Layer) 138
5.2.2 HdS Terms 139
5.3 HdS Taxonomy Axes 147
5.3.1 Introduction 147
5.3.2 Life Cycle Axis 148
5.3.2.1 System Development 148
5.3.2.2 Software and Hardware Co-Development 148
5.3.2.3 Debug and Optimization 148
5.3.2.4 Use 149
5.3.2.5 Retargeting 149
5.3.2.6 Variant or Derivative Development 149
5.3.2.7 Reuse 149
5.3.3 Run-Time and Real-Time Axis 149
5.3.3.1 Run Time 149
5.3.3.2 Real-Time 150

5.3.3.3 Communication Mechanism 150
5.3.4 Hardware Architecture Axis 151
5.3.4.1 Architecture Synopsis 151
5.3.4.2 OS Requirements 156
5.3.4.3 Architecture of Software Defined by Hardware 157
5.3.4.4 Multiprocessor Architectures 159
5.3.5 Software Layering Axis 161
5.3.5.1 Basic model 161
5.3.5.2 Layers Included in the Layered API Model 163
5.3.5.3 Control, Data, Hardware, and Software Layering 164
5.3.5.4 HdS API 165
5.3.5.5 Device Drivers 167
5.4 Conclusion 167
References 169
Index 173

ABBREVIATIONS AND ACRONYMS

ADC	Analog-to-Digital Converter
ALU	Arithmetic-Logic Unit
API	Application Programmers Interface (or Application Programming Interface)
ASIC	Application-Specific Integrated Circuit
ASP	Application-Specific Platform
BCA	Bus-Cycle Accurate
BSP	Board Support Package
CBR	Constant Bit Rate
CC	Cycle-Callable
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
DAC	Digital-to-Analog Converter (or Design Automation Conference)
DAT	Dynamic-Address Translation
DDR	Double Data Rate
DEDI	Data Exchange and Dispatcher Interface
DFG	DataFlow Graph
DFT	Design For Test (or Testability)
DFV	Design For Verification
DMA	Direct Memory Access
DRAM	Dynamic Random-Access Memory
DRC	Design Rules Check (or Checker)
DRL	Dynamically Reconfigurable Logic
DSP	Digital Signal Processor (or Processing)
DUT	Design Under Test
DUV	Design Under Verification
DWG	Development Working Group
EDA	Electronic Design Automation
EDIF	Electronic Design Interchange Format
EEPROM	Electrically Erasable and Programmable Read-Only Memory
EIA	Electronic Industries Alliance
EJTAG	Extended JTAG (Joint Test Action Group)
EPROM	Erasable Programmable Read-Only Memory
ERC	Electrical Rules Check

ESAPS	Engineering Software Architectures, Processes and Platforms for System-families
ESA	European Space Agency
ESI	European Software Institute
ESW	Embedded SoftWare
FIFO	First In First Out
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
FSM	Finite State Machine
FTP	File Transfer Protocol
FXP	Functional XML Parser or Fixed Point
GPP	General-Purpose Processor
GPRS	General Packet Radio System
GPS	Global Positioning System
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HDL	Hardware Description Language
HDVL	Hardware Design and Verification Language
HdS	Hardware-dependent Software
HLL	High-Level Language
HTTP	HyperText Transfer Protocol
HVL	Hardware Verification Language
HW	Hardware
IC	Integrated Circuit
ICE	In-Circuit Emulator (or Emulation)
I/O	Input/Output
IOCTL	IO ConTroL
IP	Internet Protocol (or Intellectual Property)
ISA	Instruction-Set Architecture
ISR	Interrupt Service Routine
ISS	Instruction-Set Simulator
JTAG	Joint Test Action Group
LVS	Layout versus Schematic
MCM	Multi-Chip Module
MMI	Man-Machine Interface
MMU	Memory Management Unit
MOC	Model of Computation (or Model of Concurrency)
MPEG	Moving-Picture Experts Group
NMI	Non-Maskable Interrupt
OFLT	OFf-Line Test
OMT	Object Modeling Technique

OS	Operating System
OSCI	Open SystemC Initiative
PBA	Printed Board Assembly
PCB	Printed Circuit Board
PBD	Platform-Based Design
PBDS	Platform-Based Design System
PDA	Personal Digital Assistant
PLA	Programmable Logic Array
POSIX	Portable Operating System Interface
PPP	Point-to-Point Protocol
PROM	Programmable Read-Only Memory
PSoC	Programmable System-on-Chip
PV	Programmers View
PVT	Programmers View with Timing
RAM	Random Access Memory
RASSP	Rapid Prototyping of Application Specific Signal Processors
RISC	Reduced Instruction-Set Computer
RMA	Rate Monotonic Analysis
ROM	Read-only Memory
RTL	Register Transfer Level
RTOS	Real-Time Operating System
RTTI	Run-Time Type Information
RTWG	RASSP Terminology Working Group
SDF	Synchronous DataFlow
SEI	Software Engineering Institute
SLD	System-Level Design
SLIP	Serial Line Interface Protocol
SoC	System-on-Chip
SRAM	Static Random-Access Memory
SW	Software
TF	Technology Foundation or Timed Functional
TLB	Translation Look-aside Buffer
TLM	Transaction-Level Modeling (or Model)
UART	Universal Asynchronous Receiver/Transmitter
UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunications System
UTF	Untimed Functional
VBR	Variable Bit Rate
VC	Virtual Component
VCI	Virtual Component Interface
VHDL	VHSIC Hardware Description Language

VHSIC	Very High Speed Integrated Circuit
VLIW	Very Long Instruction Word
VM	Virtual Memory
VSI	Virtual Socket Interchange
VSIA	Virtual Socket Interface Alliance
VoIP	Voice over Internet Protocol
VoP	Voice over Packet
XML	Extensible Markup Language
WCET	Worst-Case Execution Time

CONTRIBUTORS

The editors would like to apologize to anyone who contributed to this work, but whose name is not in the following lists.

RASSP Taxonomy Working Group

Carl Hein
Dr. Vijay Madisetti
Arnold Bard
Robert Klenke
Mark Pettigrew
Dr. Geoffrey Frank
Randy Harr
Todd Carpenter
Allan Anderson
J.P. Letellier
Dr. Greg Peterson
Dr. Perry Alexander
Anthony Gadient
Paul Kalutkiewicz

VSIA Contributors

Major Contributors:

Bob Altizer
Tom Anderson
Wael Badawy
Brian Bailey
Robin Bhagat
Neil Bryan
Steve Burchfiel
Ramesh Chandra
Larry Cooke
Mark Genoe
M M Kamal Hashmi
Anssi Haverinen
Lisa Hsu
Yaron Kashai
Chris Lennard
Srinivas Madaboosi
Roel Marichal
Grant Martin
Mehdi Mohtashemi
Carlos Oliver-Yebenes
Stephen Olsen
Peter Paterson
Frank Pospiech
Richard Raimi
Mandayam Srivas
Colin Tattersall
Jim Tobias
Kumar Venkatramani

Other Contributors:

Chris Adenyi-Jones
Bert Aerts
Jeff Barkley
Mike Bartley
Shay Ben-Chorin
Robert Birch
Ted Lehr
Adriana Maggiore
Daniel Martin
Jean Mermet
Dennis Mitchler
Richard O'Connor

Sherri Bishop
Thomas Borgstrom
Mark Buchanan
Annette Bunker
Jean-Paul Calvez
David Courtright
Gjalt de Jong
Bryan Dickman
John Emmitt
Pierangelo Garino
Olivier Giaume
Dave Goldberg
John Goodenough
Martin Gregory
Lisa M. Guerra
Mariella Guerricchio
Michael Hale
Thomas Harms
Takashi Hasegawa
Frank Hellwig
Merrill Hunt
Takahide Inoue
Mike Kaskowitz
Holger Keding
Peter Klapproth
Klaus Kronlöf
Joachim Kunkel
Kun-Bin Lee
Mukund Patel
Michael Payer
C.Y. Pei
Carl Pertry
Mark Peryer
Ian Phillips
Andrew Piziali
Sreeni Rao
Fabio Ricciato
G. David Roberts
Larry Rosenberg
Jeff Russell
Heinz-Josef Schlebusch
Sandeep Shukla
Juha-Pekka Soininen
Richard Stolzman
Ji-Suhn Suh
Kari Tiensyrja
Lee Todd
Maura Turolla
Ivo Vanderweerd
Antonio Varriale
Ralph von Vignau
Glenn Vinogradov
Janet Wedgwood
John Wilson
Parviz Yousefpour
Frits Zandveld

Companies and Organizations Represented by Contributors:

0-In Design Automation
Alcatel
Analog Devices
ARM
ARPA
BASYS Consulting
Beach Solutions
BOPS
Cadence Design Systems
Easics
Nortel
PalmChip
Philips Semiconductors
Qualis Design
Rockwell
RTI Center for System Eng.
SCRA
SES
SIPAC
Sonics

ECSI
Fujitsu
Elixent
Georgia Inst. of Technology
Honeywell
Hewlett Packard
IBM
Infineon
Intel
IRESTE, University of Nantes
Lockheed Martin
Mentor Graphics Corp.
Metamorfos Systems
MIT Lincoln Labs
Motorola
National Semiconductor
Naval Research Labs
NCTU
Sony
ST Microelectronics
STARC
Symbios Logic
Synopsys
Telecom Italia Lab
Toshiba
TransEDA
University of Calgary
University of Cincinnati
University of Tennessee
University of Texas, Austin
US Army
Verisity Design
Verplex
Virginia Tech University
VSIA
VTT Electronics

PREFACE

Yesterday's boards have become today's "Systems-on-Chips," consisting of specific architectures with embedded software components that can cooperate with dedicated co-processors. Due to the costly integration, processing and testing phases of the design cycle of such system chips, the modeling of the complete system or of specific aspects/components at various levels of abstraction is key. Moreover, high-level models allow us to specify and verify the system requirements, to analyze, explore, compare, and select different components of the system, and to explore several architectural choices. An essential element for efficient design practice is the capability to extensively re-use existing blocks or functions.

The goal of system, or high-level, models is to allow the user to evaluate and select the various components that are to be used for the System-on-Chip (SoC). Evaluation within the system environment, trade-off analysis, and subsequent decisions on items such as bandwidth, function, code size and performance can be determined within this environment in the context of the overall SoC specification.

However, meeting these system design challenges requires the unambiguous transfer of design information and communication about modeling modes between developers and providers. To address the need for conventions in modeling and terminology, this book is a collection of four taxonomies that were developed in a number of stages. This effort has included the work by a number of organizations, such as the Open SystemC Initiative (OSCI). Prior to this effort, the participating companies of the Virtual Socket Interface Alliance (VSIA) System Level Design (SLD) Development Working Group established several of the underlying taxonomies, of which the modeling taxonomy was based on the earlier Rapid Prototyping of Application Specific Signal Processors (RASSP) taxonomy and terminology efforts.

The modeling taxonomy contains an extended definition of a precision scale for the taxonomy, together with an elaborated classification of the different models used in design, implementation and verification at all levels, which are classified in system, architecture, hardware and software specific models. Most recently, multiple taxonomies covering different aspects of the design and verification process have been brought together and unified so that this single collection can cover a much larger portion of the whole space.

The underlying idea is that the design community, which includes system designers, software developers, product engineers, and hardware and verification design teams, will agree on a common acceptable nomenclature and classification of models, tools and techniques in use.

Where conflicting meanings exist in the different communities involved, the taxonomies in this collection endeavor to either choose the most common meaning or to synthesize an enveloping definition. Where this process is incomplete or impractical, all the relevant definitions and their context will be given, along with a recommended context-free default meaning.

CHAPTER 1 INTRODUCTION

1.1 What is a Taxonomy?

The word *taxonomy* comes from the Greek *taxis*, meaning arrangement or division, and *nomos*, meaning law. Thus taxonomy is the science of classification according to a pre-determined system. The Webster online dictionary [WEB] defines taxonomy as:

> *A systematic arrangement of objects or concepts showing the relations between them, especially one including a hierarchical arrangement of types in which categories of objects are classified as subtypes of more abstract categories, starting from one or a small number of top categories, and descending to more specific types through an arbitrary number of levels.*

Perhaps the most famous taxonomy is that created by the Swedish scientist Carl Linnaeus, who managed to create a classification for all living things. That classification, first published in 1735, is still in use today although with numerous modifications. In 1966, Flynn created a taxonomy for computer architectures that categorizes computers based on their streams of information. Many variants of this have since been created.

1.2 About this Collection of Taxonomies

Differing terminology has created confusion among Electronic Design Automation (EDA) tool vendors, component providers, semiconductor companies, and system houses. Some organizations use many common modeling terms with divergent meanings, while others use different words to describe the same type of models. Without a common language, the complete IC design community cannot effectively communicate, and the evaluation, selection, and validation of models and designs will be incompatible, and more difficult than necessary.

1.3 The History of the Taxonomies

Taxonomies such as the ones in this book do not just get created by a single group of people; instead they evolve over time, with each group adding a further level of refinement or understanding about the problem. At the same time, the industry moves on and what may have worked in the past needs to be updated to address new situations or emerging technologies. These taxonomies are no different and have already been through a number of significant stages of development and levels of refinement. The production of this book is just one of those stages, and will no doubt not be the last. With its publication, these taxonomies will become available to a much wider audience than in the past, and that is likely to lead to the identification of any number of possible problems, suggestions for improvement, refinements and corrections.

As designers and developers use these taxonomies, the editors expect that the definitions may be found inadequate in some respects, or not fully in line with evolving design practice. *We encourage feedback in order to improve the taxonomy definitions*. For the latest updates to this document, and discussion about any of the definitions given in any of the four taxonomies, the reader should go to the www.edataxonomy.com Web site.

1.3.1 Early Model Taxonomy Work

In the academic community, a number of model definition approaches were proposed and considered for use in the original formation of the precursors for the modeling taxonomy. Three of those model definition approaches (shown in Table 1-1) were examined and compared, feature for feature. The RASSP taxonomy (RTWG) is also shown in Table 1-1; it became the basis for the second stage of development of this taxonomy–the VSIA effort.

The Ecker and Madisetti spaces share two axes of comparison, while their remaining axes do not directly correspond. Both have an axis for time resolution and a second axis representing the resolution of data values in a model.

Ecker calls the second axis "Value," while Madisetti calls it "Format." The Y-chart's "Functional Representation" axis expresses some information that is similar to the value-format axes. However, the Y-chart's Functional-Representation axis does not exactly correspond to the value-format axes because it contains information about functionality as well.

The third axis of the Ecker cube is similar to the "Structural Representation" axis of the Y-chart but has no corresponding axis in the Madisetti case. (The latter situation arises intentionally.)

None of the remaining axes of the taxonomies directly correspond. The Y-chart seems limited to only the logic level. None of the taxonomies appeared to have directly addressed the hardware/software co-design aspect.

Source - Taxonomy	**Axes**						
Gajski and Kuhn: Y-chart			Structural Represen-tation	Functional Represen-tation	Geometric Represen-tation		
Ecker: Ecker Cube	Timing	Value	View				
Madisetti Taxonomy	Timing	Format				Value	State
RASSP RTWG Taxonomy	Timing Resolu-tion	Data Value	Structural Resolution	Functional Resolution			Intern/ Extern

Table 1-1 Comparison of Prior and RASSP Taxonomy Concepts

1.3.2 First Model Taxonomy Industrial Effort

The concept of a model taxonomy for the industry was initiated by efforts of the RASSP Terminology Working Group (RTWG), which was formed on January 10, 1995, at the RASSP Principal Investigators meeting in Atlanta, Georgia, to address modeling and terminology challenges. The core working group consisted of members representing the two prime contractors, a technology base developer, the educator facilitator, and the government.

The RTWG's mission was to develop a systematic basis for defining model types and to use this basis for concisely and unambiguously defining a terminology that describes the models that are used within a RASSP design process. One crucial requirement for the basic taxonomy was that it must be useful for selecting, using, and building appropriate interoperable models for specific roles in a RASSP design process. Models are used for several purposes, which include specifying or documenting design solutions and testing and simulating proposed designs. The terminology was based on the commonly documented and applied vocabulary in the digital electronic design and modeling industry at that time, and it drew heavily from related earlier and ongoing efforts by the EIA, ESA, and the U.S. Army and Navy, and from the annals of related literature from the Design Automation Conference (DAC), VHDL International User's Forum, and text books.

Previous efforts focused on narrower domains than RASSP. RASSP spanned many domains, including parallel processing; multi-board and multi-chassis systems; software; digital signal processing; and application

functions, with strong interaction with other domains such as analog, mechanical, physical, and RF.

1.3.3 Work within the VSIA

The System-Level Design Development Working Group (SLD DWG) of the VSIA modified and augmented the previously defined terminology sets, broadened parochial definitions, distinguished overlapping definitions, equated close synonyms, removed inapplicable terms, added needed or missing terms, clarified poorly defined or misunderstood terms, and suggested new terms as replacements or synonyms to outdated terms. When appropriate existing definitions were not available for significant terms used within the VSIA community, the SLD DWG attempted to create them. At different places the definitions were illustrated by concrete examples.

Compared with the RASSP document, this taxonomy and model classification proposal was further elaborated by the working group by adding more appropriate details on the different precision scales, and by providing more concrete examples for each possible precision. In addition, several additional models were added, such as computational, architectural, and software models. Some of these extensions were adopted by the RTWG during the active period of work. Two major revisions of this taxonomy were released to the public through VSIA.

1.3.4 Extension into other Areas

The launch of the VSI model taxonomy document was highly successful and for most months was the most downloaded document from the VSIA web site. For quite a few months it exceeded the number for all of the other documents combined. Based on this success, other groups that were started within the VSIA took, as their first goal, the production of similar documents for their particular spaces. This served to ensure that all of the working group members had a consistent language when talking about the follow-on documents and standards, and served as a learning process to uncover places of contention in the industry. Three other working groups produced these documents–the Functional Verification DWG, the Platform-Based Design DWG, and the Hardware-dependent Software DWG. Perhaps the most contentious of these was the document that came out of the Platform-Based Design working group, as it showed the enormous range of opinions and definitions for platforms in the community. All four of the VSIA-developed taxonomies are contained within this collection.

1.3.5 Latest Revisions

This book provides another chapter in the life of the taxonomies. Since the time of the last major revision to the model taxonomy, a number of new organizations have emerged that have added new terms, modified the boundaries set for certain abstractions, and defined completely new levels of abstraction. This revision considers the current state of the efforts within the Open SystemC Initiative's (OSCI) Transaction Level Modeling (TLM) group. While this group had not completed its standard at the time of this book's publication, and contains two rather different ways that the standard could evolve, both of these sets of models have been included as they provide concrete illustrations of a number of the model types.

In the earlier forms of the modeling taxonomy, little attention had been paid to the variety of models of computation that are used as the basis behind many of the models described in the taxonomy. A more encompassing description of these has been added, based on classifications and analysis approaches of Axel Jantsch, KTH, Sweden.

Significant updates have been made in all four taxonomies to reflect evolution in the industry since original publication, to incorporate recent work, and to better unify style and organization.

Given VSIA's focus on design reuse, the original taxonomies were largely developed from the perspective of a Virtual Component (VC), a design block intended for reuse in multiple chips. Since most of the terminology and model definitions in the taxonomies are applicable to any design, most references to VCs have been removed in this book. However, there are a number of cases in which the distinction between a reusable design block and the entire chip in which it is integrated are important, and in these cases the term has been retained.

1.4 Taxonomy Organization

This taxonomy collection is composed of four primary sections, each one dealing with a specific aspect of the design and verification process. The four sections are:

- **Model taxonomy:** this section of the collection defines the core of the model taxonomy. It provides a definition for each of the commonly used levels of abstraction and provides examples of how these are used within the industry.
- **Functional Verification Taxonomy:** this section deals with one of the biggest challenges in electronic system development–verifying

the functionality of a reusable design in different chips, in different physical implementations, and in different development environments. There are a number of tools, techniques, and methodologies used to accomplish the functional verification of the component and the system. This section of the collection is intended to provide a classification of the various verification technologies and uniform definitions of terms used in these technologies.

- **Hardware-dependent Software:** this section deals with software, which plays an increasing role in SoC design. Therefore, reusability considerations must now address software layers as well as hardware. At the lowest level, a software layer interacts directly with the interface offered by the SoC's hardware platform. This software layer is defined to hide hardware specifics from the upper layers of software. Hardware-dependent software (HdS) can be viewed from the perspective of a software platform, hardware platform, or SoC design life cycle.
- **Platform-Based Design:** this section contains the latest understanding regarding this newly emerging area. It attempts to define the key concepts of platform-based design and their meanings, and the attributes by which platforms can be classified. Its scope is all platform-related development at any level within an SoC.

In addition, other sections of the document provide definitions for the standard vocabulary used within the industry.

1.4.1 Use of this Book

This collection of taxonomies is intended to be of use to a wide audience. First, model developers, whether third-party or within large companies, can begin by re-classifying the various models provided with their components into the categories offered by the model taxonomy, and can refer to this book for explanations. Model integrators can then begin to request models according to the taxonomy definitions. EDA suppliers and design-methodology developers at semiconductor and systems companies can begin factoring these model types into their tools and methods. They can also use the taxonomy to clarify which types of models fit into their design flows, and how.

We also hope that evolution in design practices for components and component integration into SoCs will help identify which model types are critical to the methodologies, and which models may only be peripheral to them. This will help reduce the number of models requested and required to permit efficient design, and also clarify their characteristics along the

resolution axes. Keeping the number of model types to a minimum will be a significant help in the evolution of the electronics industry toward a reuse-oriented mindset by reducing the overhead required to produce a complete reusable design package.

The final uses of this book could be for educational purposes, and in industry standards efforts based on these model types. In this domain, the clarifications offered by the taxonomies and the definitions of models should help in teaching and using a better common language, which will assist in education, and help standards groups make quicker progress. Experience in system-level design across industry, university, and government groups indicates that a large part of the initial time involved in their activities is spent trying to agree on language, definitions, and model types. This book should reduce the time spent coming to a consensus on these terms.

CHAPTER 2 MODEL TAXONOMY

2.1 Introduction

A modeling taxonomy provides a means to categorize models according to a set of attributes. The attributes should be useful in distinguishing models intended for distinctly different purposes. The taxonomy is used to establish formal definitions that are concise and unambiguous for the various model types. Descriptions and definitions for many of the terms used in this document are provided in Section 2.9, "Supporting Terms" of this chapter.

2.1.1 Taxonomy Definition

This taxonomy represents the model attributes that are relevant to designers and model users. It consists of a common set of attributes to independently describe a model's internal and external resolution. This taxonomy is based on terminology readily understood and used by designers.

The axes explicitly characterize a model's relative resolution of details for important model details. The taxonomy axes, shown in Figure 2.1, identify four model characteristics:

- Temporal detail
- Data value detail
- Functional detail
- Structural detail

The temporal and data axes are clearly orthogonal to each other, and to the other two axes. In contrast, the relationship of the functional and structural axes is not totally orthogonal, but it is useful to consider these two aspects of a model through the different attribute filters even though they may be connected in some ways.

Distinguishing between the internal and external views is important in selecting, using, and building models because it enables clarity and precision. Previous terminologies often mixed attributes, as viewed from inside a model, with similar attributes, as viewed from the model's interface boundary.

- Internal resolution references how a model describes the timing of events, functions, values, and structures of the elements that are contained within the boundaries of the modeled device.
- External resolution describes how a model describes the interface of the modeled device to other devices. The external aspects refer to the input/output (I/O) details at the boundary of the modeled device. The external details relate to how the model describes a device's interaction with devices to which it connects. External details may include timing and functional aspects, commonly referred to as protocols, as well as port structure and signal values. All of these aspects may be abstracted to various levels in a model.

Because each aspect is specified from both an internal and external viewpoint, the taxonomy effectively contains eight attributes describing a model's descriptive level–*internal attributes:* temporal, value, structure, and function resolution; and *external attributes:* temporal, value, structure, and function resolution. The *resolution* of an axis defines the expected *precision* of the models at various points on the axis–the model itself would define its *accuracy* at that resolution or precision. These three terms, as used with respect to the axes and models, can be used in similar, confusable ways, but the recommendation is that accuracy is used for models, and precision or resolution is used for axis-related measures.

This set of eight attributes does not address the hardware or software co-design aspect of a model, because it does not describe how a hardware model appears to software. Therefore, the set is augmented with a ninth axis (shown at the bottom of Figure 2.1). This axis can either represent the level of software programmability of a hardware model or the abstraction level of a software component in terms of the complementary hardware model that will interpret it. This axis is not orthogonal to the temporal, data, functional, and structural axes, but is intended to make clear, in classical software notations, the level of the model in its software aspect. Thus, it is an aggregate property that reflects, to some extent, characteristics defined along the temporal, data, functional, and structural axes. Within this model taxonomy, the software programming resolution is clearly an approximation. For a full definition of the relationship between hardware and software, it is advised that you follow the definitions given in Chapter 5, "Hardware-dependent Software."

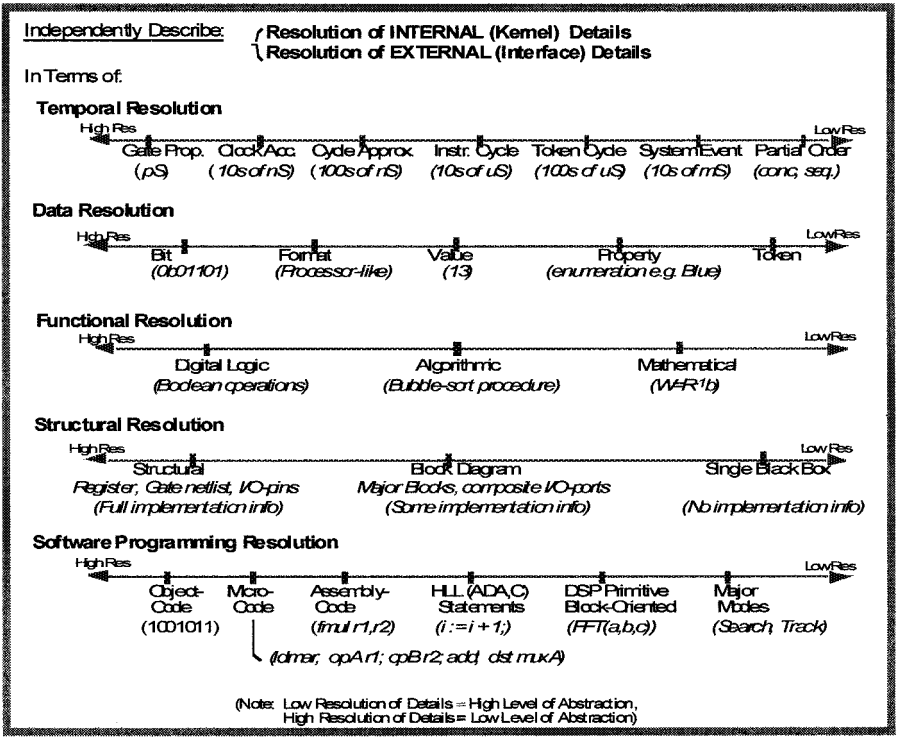


Figure 2.1 Taxonomy Axes

Although Sections 2.1 through 2.7 of this chapter define the vocabulary terms graphically to show their applicable coverage, a convenient method for specifying a particular model's information content is to use the Internal/External (temporal, value, function, structure) notation.

For example, the content of a particular RTL model could be specified as:

Internal(temporal=Gate Propagation, data=Bit,
function=Digital Logic, structure=register),
External(temporal=Clock Accurate, data=Bit,
function=Digital Logic, structure=I/O Pins),
SW-Program(Programming-Level=Assembly-Code)

For contrast, an example of a particular algorithm model could be specified as:

Internal(temporal=System Token, data=Value,
function=Algorithmic, structure=none),
External(temporal=none, data=Value,

function=none, structure=none),
SW-Program(Programming-Level=none)

2.1.1.1 Temporal Resolution Axis

The temporal resolution axis represents the degree of accuracy of events that are modeled along a time scale, or in time. There are several levels of precision implied on this axis:

- **Partially ordered event accurate**: at this level of precision, events have a complete or partial ordering relationship in terms of their starting and finishing, without specifying precisely when those events start and finish in terms of any temporal units. Thus, the precedence of events is indicated. This level of precision is common in dataflow analysis. The partial ordering means that independent threads of event occurrence may only be ordered within the thread and not have any ordering relationship between the threads.
- **System event accurate**: at this level, start and end times of major system functions are indicated in some units that may represent thousands or millions of "clock" cycles.
- **Token cycle accurate**: (also may be called *data cycle*) at this level, which especially applies in periodic and dataflow kinds of systems, precision of events is defined in terms of the regular progress of data and control tokens that "flow" from one functional processing unit to the next. For example, an image processing system may process, in a regular way, X frames of image data per second, and the periodic processing of image frames (tokens) defines a periodic "clock" interval: frame 1, frame 2, frame 3, and so on.
- **Instruction cycle accurate**: at this level, events are specified in terms of the processing of an instruction stream or transaction. This is more precise than the token cycle, in that several operations may be required to process one data token.
- **Cycle-approximate accurate**: at this level, approximate cycle counts are available for each operation, transaction or message that is processed, in terms of a system "clock." For example, a processor model may not completely model every aspect of the processing of an instruction stream such as the modeling of pipelining and cache effects. This means that cycle counts produced and accumulated may not be completely accurate, and thus the use of the term "cycle-approximate."
- **Cycle-accurate**: at this level, accurate cycle counts are available for each operation, transaction or message that is processed, in terms of a regular system clock. Events occurring during the processing of

such operations are indicated with exact precision as to which clock cycle they occur in.

- **Gate propagation accurate**: at this level, event starting and ending times are defined in terms of precise time units within clock cycles–for example, nanoseconds (ns) or picoseconds (ps)–not just on clock cycle edges or boundaries. The accuracy of these event timings depends on the degree of accuracy of the circuit level and the interconnect models used to predict them.

The concept of the "delta" cycle, which enables instantaneous concurrency, exists for all levels of abstraction. This is sometimes referred to as a zero-time sequence. Delta-cycles are particularly used in discrete event simulators such as HDL simulators although also used in many other simulation models and tools.

2.1.1.2 Data Resolution Axis

The data resolution axis represents the precision with which the formats of values are specified in a model. The contents of a register could be described at these levels of resolution, from high to low:

- Binary (for example, 0b111)
- Signed integer (for example, -1)
- Enumerated (for example, blue)

Note that resolution is analogous to precision, as distinguished from accuracy. Each representation is equally accurate; however, the first case resolves the value closer to the form actually contained in the target device. The more abstract the representation of a value, the fewer implementation details are resolved.

The different precision items on this axis are as follows:

- **Token**: The token precision level is the highest abstraction level for data representation, containing no implementation (structure, size, values, and so on) details at all. The amount of information it contains is completely unspecified.
- **Property**: An example of a property precision is an enumeration: you decide that the datum "color" will have the properties "red," "blue" and "green". You may do this via an enumeration (which gives you some ordering properties), or you can do it in some other way (such as a string plus some object methods). This data format includes user-specified data formats based on previously defined or standard data formats (at a lower abstraction level).
- **Value**: At the value level, there are no implementation details. Although the value may be an integer or real, the details as to how

this is represented, such as fixed point floating point, are not described at this level.

- **Format (Processor-Like):** A more detailed data precision level is the processor-like data format, for example, a "big-endian" or "little-endian" formatting of addressing the actual byte orders. This level includes such format concepts as whether a value is fixed-point or floating-point (with fixed mantissa and exponent), and so on.
- **Bit Logical:** Here, the final representation can be used on a bit-by-bit basis. For each bit, its logical value may include binary and multi-valued representations. More details can be added in the possible values that each bit can represent (such as 1, O, X, Z, and so on).

A *composite* is a representation that is formed by a combination of types.

2.1.1.3 Functional Resolution Axis

The functional resolution axis represents the level of detail with which a model describes the functionality of a component or system. A digital filter component could be represented by these levels of resolution, from high to low:

- **Mathematical Relationships:** At this level of abstraction, the functionality is represented as a set of mathematical equations, without sequencing, except for that defined by the rules of precedence for arithmetic equations.
- **Algorithmic Processes**: At this level, the selection of the algorithm will be made, such as a bubble-sort procedure. There are currently no details on the way this algorithm will be implemented. The algorithmic precision includes sequencing, since the ordering of operations and control flow is a key criterion for selection. The algorithm may be expressed in a number of ways, including decomposition into major functional portions linked in a network (for example, a dataflow diagram, or a process network). However, the structure of this decomposition does not necessarily imply anything about the implementation of a function: its implementation may be done in a completely different way. Any algorithmic decomposition here may be purely for convenience in defining the function.
- **Digital Logic/Boolean Operation**: At this level, the functionality will be specified at the level of Boolean operators (AND, OR, NOT, and so on). Thus the structural content and resolution of this type of model would probably be high.

2.1.1.4 Structural Resolution Axis

The structural resolution axis represents the level of detail with which a model describes how a component is constructed from its constituent parts. An integrated circuit could be represented at these levels of resolution, from high to low:

- No Implementation Information:
  - No structural information–one large block
- Some Implementation Information:
  - Connection of large blocks, such as an ALU and register files
  - Connection of computer networks
- Full Implementation Information:
  - Connection of simple units, such as logic gates
  - Connection of composite units, such as flip-flops
  - Connection of more complex units, such as registers and multipliers; at this level the block diagram has been further expanded into basic operators such as adders, multipliers, shifters, and so on, or even into more detailed granularity such as logic gates

Structural resolution is not limited to the physical implementation of integrated circuits described in the previous examples. As in the definition, a model can be built out of constituent parts in non-physical ways. A design block that is implemented entirely as software is built out of sub-components such as procedures and processes linked in complex networks.

A dataflow function, implemented by mapping to dedicated hardware blocks, may have a one-to-one relationship between a functional decomposition and the basic implementation structures that realize each function.

The concept of black-box and gray-box models has some similarities to the structural resolution axis. A black-box model hides all the internal structure within the component. This is identical to the resolution level "no implementation information" on the structural resolution axis. A gray-box model is more complex. Some of the internal structure within the component is exposed, but the model hides most of the detail of combinational logic. Thus, a gray-box model contains all sequential devices, and represents combinational logic between sequential device pairs by timing arcs. This is a more complex version of structure than is contained on the current structural resolution axis; thus, these concepts are not completely unified.

2.1.1.5 Software Programming Resolution Axis

The software programming resolution axis is the granularity level of software instructions that the model of a hardware component interprets in executing target software. For instance, a network-performance model only interprets instructions on the level of dataflow primitives, such as matrix invert, vector multiply, or Fourier transform. Such primitives represent hundreds of lines of source code, but are interpreted as a single instruction in terms of a time-delay by a network performance model. An instruction-set-architecture (ISA) model interprets individual assembly (or object code) instructions. In this sense, the ISA model is programmable at a much finer granularity, or higher precision, than the network-performance model.

At the lowest extreme, a model of a microcode programmable processor is programmable at an even lower level of granularity than the ISA model because it allows control of individual register and multiplexer structures within the device during execution of an assembly-level instruction. Software design components or non-programmable models are at the opposite extreme because neither interprets programmable instructions.

The software programming resolution axis in Figure 2.1 represents the granularity of software instructions that the model of a hardware component interprets in executing target software. In Chapter 5, "Hardware-dependent Software," a complete taxonomy of software utilizing multiple axes is defined. The view in Figure 2.1 is a simplification of the complete taxonomy and, thus, should only be used as an approximate guide. A programmable device could be represented at these levels of resolution, from high to low:

- **Major Modes:** At the highest precision level, the software is specified in terms of major working modes, such as searching, tracking, initialization, and so on.
- **DSP Primitive block-oriented:** At this level, the software is expressed as a block or function call and its parameters, for example, an FFT with parameters a, b, and c specified as FFT(a,b,c).
- **High-Level Language**: The software is specified at this precision level in terms of high-level language statements. Examples of such languages are C, C++, ADA, and Java.
- **Assembly Code:** A more detailed level is the assembly code level, which is a symbolic language that can be translated later into microcode and object code. Assembly code is usually the result of the usage of a compiler, but it can also be written manually.
- **Microcode**: The assembly code can be translated in even lower level of instructions, called microcode instructions, which are a representation of an set of control signals that are active on a given clock cycle, as well as the next micro-instruction to be executed.

- **Object Code**: The lowest level of precision contains the translation into binary code.

2.1.2 Internal/External (Interface) Concept

To better understand the internal/external concept, consider two views of an integrated circuit chip.

When viewed from outside, or externally, we observe only the structure and behavior of the pins (for example, how many pins there are, and what values they have when driven with various stimuli). But we cannot observe any details about how the chip is implemented inside the package, or internally, as in Figure 2.2.

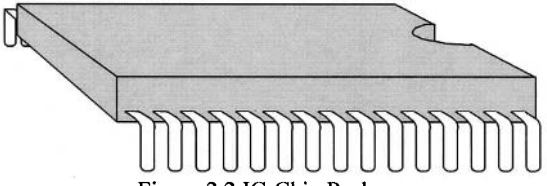


Figure 2.2 IC-Chip Package

In contrast, we can imagine seeing an internal view if we were to pop the lid off the IC package as shown in Figure 2.3.

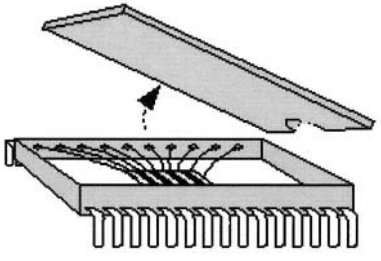


Figure 2.3 IC-Chip Internals

Notice that now we can see some detail about how the chip's insides, or internals, are implemented.

The external structure and behavior is the structure and behavior of the externally observable features, which in this case is the structure of the externally viewable ports or pins. Like the internal design, the external properties of a component can be viewed at many different levels of abstraction.

For example, Figure 2.4 depicts an abstract model of the external pins (interface) of a chip. The external implementation detail is resolved as two

signals (Data and Control) that are of an abstract type, integer. They are abstract because they do not reveal the bit-level implementation detail.

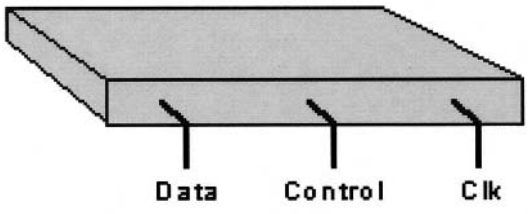


Figure 2.4 IC-Chip External Pins

A less abstract model of the external interface of the component could show the actual bit-level implementation detail of the signal ports, as shown in Figure 2.5.

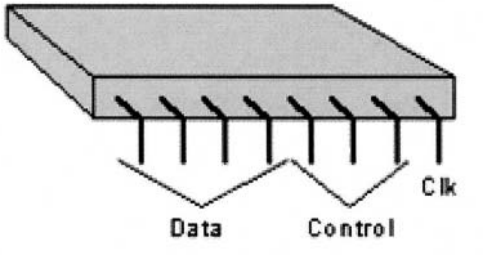


Figure 2.5 Signal Ports

This more detailed view shows the hand-shaking lines and the data port resolved as individual pins at the bit level.

The internal/external partition concept simplifies the design process. A traditional method for managing complexity in a system is to use *divide-and-conquer* methods. In this approach, a system is divided by partitioning it into separate components. These components can, in turn, be further subdivided. This leads to the familiar hierarchical models used in most design methodologies.

However, this partitioning will not help the design task if, at each level of hierarchy, the designer has to consider each component's internal detail and its sub-components. What is needed is to be able to consider only a component's externally visible specification and not its internal detail. Most methodologies and languages try to do this by having separate structures for the *interface* and the *body* of a component. Users of a component only need to see its interface. Thus, an interface specification or interface model is the description of the externally visible part of a component.

2.1.3 Note on Structure/Behavior/Interface Concepts

Another way of depicting the view of a model is shown in Figure 2.6 through Figure 2.9. These diagrams distinguish:

- Interface model (no internal details)
- Behavioral model (internal details described behaviorally)
- Structural model (internal structure described)

Figure 2.6 depicts an interface model. Notice that it contains details about the interface, or external ports, but contains no information about the internal implementation. Some level of external structure and data values can be observed, as well as some level of port function and timing response to interface activities. For example, the interface model can specify the data exchange and communication protocol, as implied in Figure 2.7.

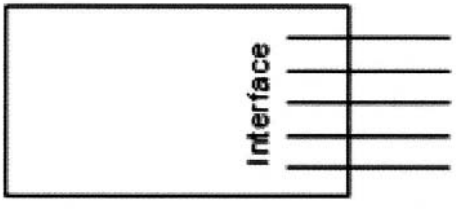


Figure 2.6 External Interface

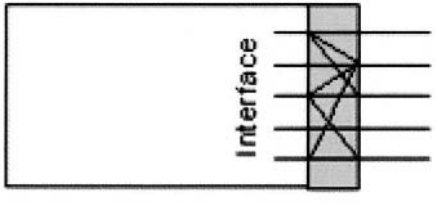


Figure 2.7 Internal Interface

In contrast, Figure 2.8 depicts a behavioral model of the same component. Notice that this model contains information about the internal data values, functions, states, and timing aspects of the component, but no information about how the internal structure is implemented. Therefore, the internal view is said to be represented behaviorally.

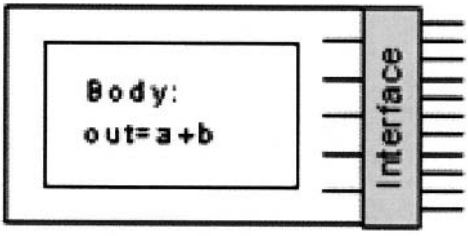


Figure 2.8 Behavioral Model

In contrast to this model, Figure 2.9 depicts a model that describes the internal structure of the component to some level of detail. Remember that structure is inter-connection information. Notice that this diagram shows a decomposition into internal blocks. Because it shows how the blocks are connected to each other, at some level of abstraction, it is called a structural model, or internal structure.

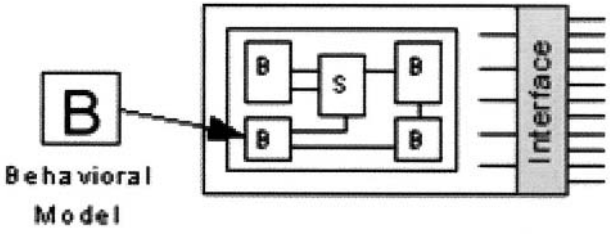


Figure 2.9 Internal Behavioral Model

The internal blocks of a structural model can either be described behaviorally, or can themselves be further decomposed structurally. If behaviorally described blocks exist at the bottom (leaf level) of a structural hierarchy, then the model can be simulated. Note that the behavior of such a composite model is provided by the underlying behavioral blocks, not completely by the structure. The structural descriptions merely provide the means for combining separate behavioral pieces. A different behavior would be exhibited if the underlying behaviors were changed.

If behavioral blocks do not exist at the bottom (leaf level) of a structural hierarchy, then the model is a purely structural model. No behavior can be inferred if the behaviors of the underlying blocks of a structural model are unknown.

The level of abstraction of the internal view depends on the level of implementation details. The structure could be described abstractly as the interconnection of high-level blocks, or concretely as the interconnection of logic gates. Independently, the timing and functional abstraction can be described for the high-level blocks abstractly as coarse events or concretely

as specific times. They can be described for the gate-level model abstractly as the stable per-clock Boolean values, or all switching transitions can be resolved to picoseconds (intra-clock events) and signal levels.

2.1.4 Additional Attributes

In addition to the precision axes described earlier, the inclusion of additional attributes could be considered. For instance, the temporal resolution and data resolution axes could specify an accuracy aspect as a percent tolerance and whether a model describes actual, minimum, typical, or maximum values. A completeness aspect could also be considered that would specify the portion of functionality or particular functions that the model describes or excludes from the model. The accuracy and completeness aspects would accompany the axes in the same way that the internal/external aspect does.

2.1.5 Vocabulary

Words represent concepts and allow us to share and communicate ideas. Unambiguous communication requires not only that we have a common mapping of words to concepts, but that we have the right set of words to accurately describe the concepts we are dealing with. Developing, agreeing to, and using a concise common terminology are therefore vital to achieving the goals of this book.

The development of an efficient vocabulary, which assigns a minimal set of words to the appropriate concepts, is an orthogonalization process. The process develops a set of terms that represent all of the concepts to be distinguished. Separate words are selected for distinct concepts. Words for classes of concepts are selected to represent useful generalizations.

In defining the vocabulary terms, attempts were made to defer to the general English meaning of words as defined by the Webster's New Collegiate Dictionary [WEB] so that outsiders and newcomers may be more likely to rapidly understand and adopt the terminology.

Some terms may have multiple meanings (due to historic or domain overloading) that can be differentiated by context. We try to recognize and define each of these terms.

To avoid the problem of vague or circular definitions, a heavy emphasis was placed on providing examples to accompany the definitions. These examples should provide a level of understanding and concreteness to any discussions regarding the terms. The examples also tend to associate the terms to their intended uses and domains. To reduce the tendency of

examples to limit or over-constrain the definitions, a range of typical and extreme cases are given and identified wherever possible.

To further avoid ambiguous definitions, attempts were made to eliminate definitions based purely on relative terms, such as "high," or "abstract," since their interpretation would be subject to one's point-of-view or experience. Instead, definitions should be based on absolute, concise, and testable statements, with special emphasis on differentiating related terms (such as "software/hardware/firmware").

Reaching the much-needed consensus on terminology requires compromise from everyone. The original contributors to this work and editors of this book have made a conscious attempt to borrow heavily (and somewhat evenly) from everything written by the community. All terms relative to the taxonomy axes described in this section are described in the following sections of this chapter. Although some terms may span a range of abstraction levels, a given model instance describes information at one specific level within the span. The remaining Sections 2.2 through 2.8 list the vocabulary terms and their definitions.

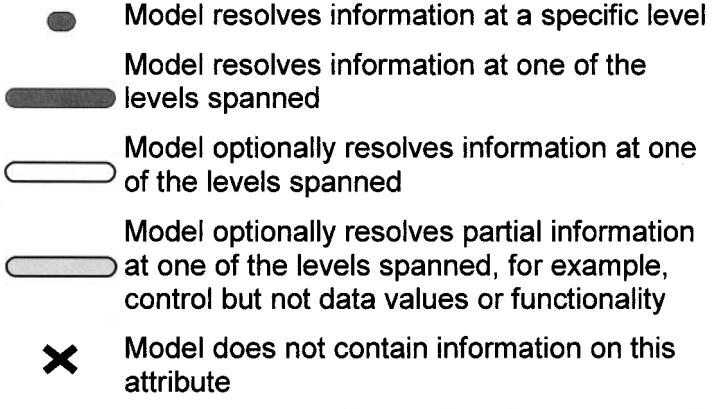

Model resolves information at a specific level
Model resolves information at one of the levels spanned
Model optionally resolves information at one of the levels spanned
Model optionally resolves partial information at one of the levels spanned, for example, control but not data values or functionality
Model does not contain information on this attribute

Figure 2.10 Symbol Key

2.2 General Modeling Concepts

The following sections contain definitions of concepts that are pervasive across many types and levels of models. The modeling concepts are divided into three groups:

- Primary Model Classes:
  - Functional Model

- Behavioral Model
- Structural Model
- Interface Model

- Specialized Model Classes:
  - Performance Model
  - Hybrid (Mixed-Level) Model
- Computational Model Classes:
  - Dataflow-Graph Model
  - Other Models

2.2.1 Primary Model Classes

All models can be described in terms of one or more of the three primary aspects–behavioral, functional and structural–combined with an interface model.

Pure functional models are timeless algorithmic models. Behavioral models add time to the function, and structural models build up models from other models. Interface models exist to separate the specification of internal function in a model from the specification of its externally visible part, such as the communications protocol it uses.

The primary model classes are not specific to hardware or software, and they can be used for either. They can also be used for the system level (unmapped to hardware or software).

A functional or behavioral model may be composed of a number of smaller functional or behavioral models linked together in a network (for example, a dataflow decomposition, a process network, or multiple threads in a programming language). However, this "structural" decomposition of such models need not imply anything about the implementation of that function or behavior in the physical or software domains.

Recently the Open SystemC Initiative (OSCI) has been working on a set of model definitions that all operate at what they call the transaction level [DON 04], [DONBR 04]. They define the transaction level as being any abstraction above the RTL model but they do not provide any guidance as to whether a transaction is an atomic entity in terms of the data being transferred or not. It thus describes a concept rather than anything specific. At the same time, they have defined a specific kind of transaction called a transfer, which implies a level of timing accuracy for the transaction. While these are not very useful definitions by themselves, they are identifying a number of modeling stages which allow a flow to be defined starting with a high-level functional model, and refining this in a number of intermediate models down to a specific implementation.

Perhaps more importantly, SystemC provides for explicit separation of the function and interface, allowing each of them to be refined somewhat independently of the other. For many model definitions, they have not separated these concepts when they provided names to the various models, and this could create confusion when the abstraction level of the model internals and its interface do not match.

In the following definitions for model types, the corresponding models identified by the SystemC Transaction Level Modeling (TLM) group will be identified. It should be noted that this group has not yet finished its work and, within the group, there is some level of disagreement on the model levels that should be defined. In addition, some of the models under consideration do try to tie together the abstraction of the model and some specifics about the communications supported by the model. While this creates very highly specific types of models, direct applications for these models can be defined.

2.2.1.1 Functional Model

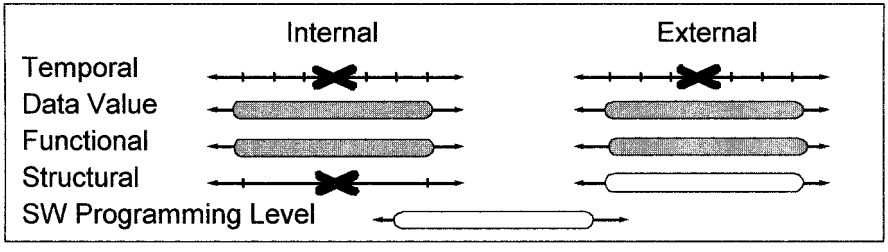


Figure 2.11 A Model without Timing

A *functional* model describes the function of a system or component without describing a specific implementation. A functional model can exist at any level of abstraction, depending on the precision of implementation details. For example, a functional model can abstractly describe a signal-processing algorithm, or it can be a less abstract model that describes the function of an ALU for accomplishing the algorithm. The precision of internal and external data values depends on the model's abstraction level. The functional model does not specify any timing other than that implied by dependencies in the function. So, for example, a functional model expressed as a series of input-output relationships with no intermediate steps has no temporal specification information at all.

A functional model can also be more narrowly defined in the mathematical sense. That is, a functional model can be described in terms of mathematical functions, which define mappings from a subset of the set of

interface variables (the input domain) onto another subset of the set of interface variables (the output domain). A partial functional model is one in which the union of these two subsets of interface variables is not equivalent to the complete set. If the two subsets are disjoint, then the first are pure "inputs" and the second are pure "outputs" of the functional model. Models that include time in either domain are more general and fall into the behavioral model definition provided in the next section.

In the SystemC TLM world this is identified as the *untimed functional* (UTF) model. The SystemC TLM definitions have also provided some hints on the external view of the model, by stating that communications occurs in the form of message passing and is point to point in nature. Communications is usually blocking, although there may be a FIFO of arbitrary depth placed in the communications channel.

Another variation of this model is the *programmers view* (PV) model. It shares the same key characteristics as the UTF model, but it does define that the functional objects that are exposed by the interfaces be modeled at a bit-true level. These models are intended to allow embedded software developers to verify their code on a model of the target platform. This is shown in Figure 2.12.

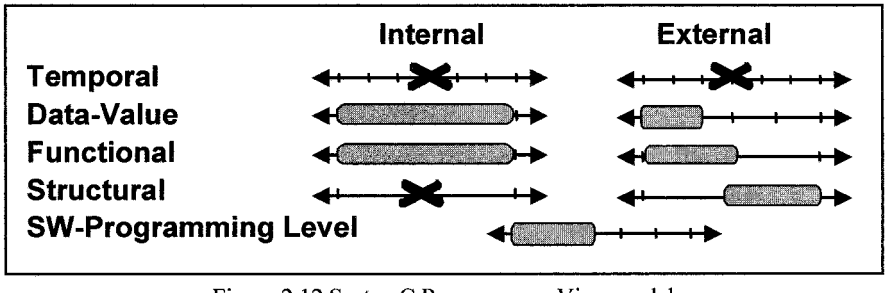


Figure 2.12 SystemC Programmers View model

2.2.1.2 Behavioral Model

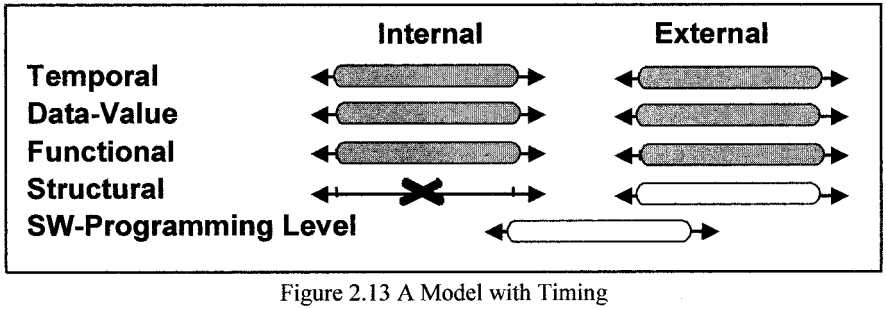


Figure 2.13 A Model with Timing

A behavioral model describes the function and timing of a component without describing a specific implementation. A behavioral model can exist at any level of abstraction. Abstraction depends on the precision of implementation details. For example, a behavioral model can describe the bulk time and functionality of a processor that executes an abstract algorithm, or it can be a model of the processor at the less abstract instruction-set level. The precision of internal and external data values depends on the model's abstraction level.

The SystemC TLM presents a modification to the behavioral model, called the *programmers view with timing* (PVT). While internally the same as a behavioral model, the characteristics of its interface are closely defined. Time details are provided along with bit-true data types, but the structure of the interface may still be abstract. They also define another close variant of this which is called the *transaction layer* (or timed functional model). The only difference between these two models is that the timed functional model has a looser definition for timing, allowing for estimated delays. Figure 2.14 shows the composite of these two models.

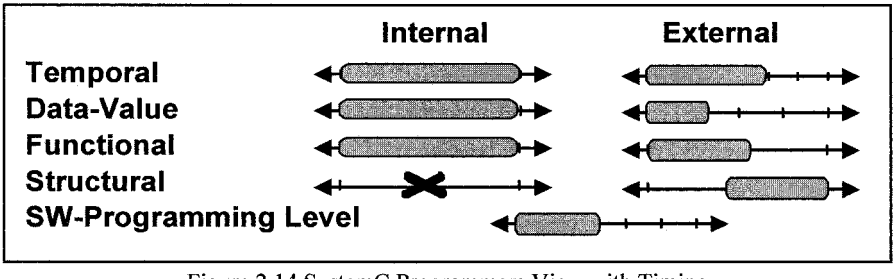


Figure 2.14 SystemC Programmers View with Timing

2.2.1.3 Structural Model

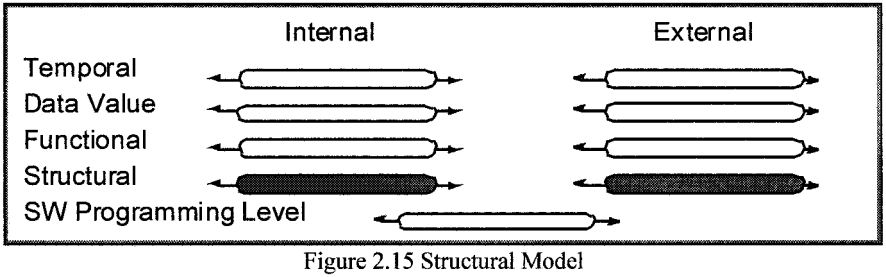


Figure 2.15 Structural Model

A *structural* model represents a component or system in terms of the interconnections of its constituent components. These components can be

structural, functional, or behavioral. So the hierarchy can reflect, for example, the organization of a set of software modules, the physical organization of a specific implementation, or the communication topology of a set of concurrent processes. A structural model that describes the physical structure of a specific implementation specifies the components and their topological interconnections. Simulation of a structural model requires all the models in the lowest (leaf-level) branches of the hierarchy to be behavioral or functional models. Therefore, the effective temporal, data value, and functional resolutions depend on the leaf-level models. A structural model can exist at any level of abstraction. Structural resolution depends on the granularity of the structural blocks.

For a set of software modules (objects, threads or tasks), the linkage model (such as the nature of the interconnections) is fundamental to the behavior or function of the resulting sub-system. Similarly, for a set of hardware structures, the linkage model (such as a communications network) is of fundamental importance to deriving the overall system behavior or function.

2.2.1.4 Interface Model

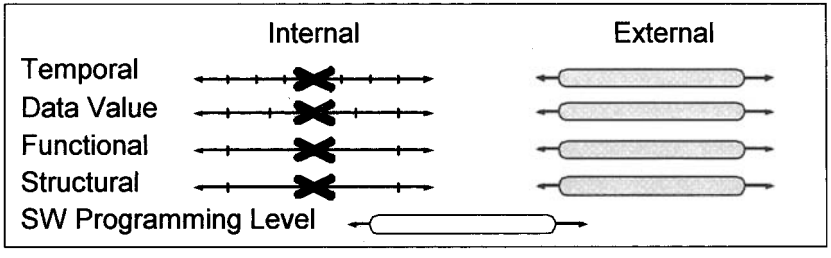


Figure 2.16 Interface Model

An *interface* model is a component model that describes the operation of a component with respect to its surrounding environment at some level or levels of abstraction.

The terms "bus functional" and "interface behavior" have also been used to refer to an interface model. The more general "interface model" name is preferred to the anachronistic bus functional term because bus functional models usually represent lower levels of abstraction and hence are a subset of interface models.

Interface models provide or capture information only on the external axes of any of the model types defined in this taxonomy. Therefore, any of the model types in this taxonomy can have an interface model.

In an interface model, some level of external structure and data values may be observed, as well as a level of function and timing response to interface activities. That is, the interface model can specify the data exchange, dependencies and communication protocol of the component's externally visible features.

In interface models, the external connective points (such as ports or parameters), functional constraints, and timing details of the interface are provided to show how the component exchanges information with its environment. An interface model contains no details about the internal structure of the component, function, data values, or timing, other than what is necessary to accurately model the external interface behavior. External data values are usually not modeled (except for constraints) unless they represent control information. An interface model may describe a component's interface details at any level of abstraction, such as message tokens or bit accurate.

A complete interface model for a component would provide a useful black-box specification for that component, since it would specify the data types and communication protocols used by the interface at multiple levels of abstraction. All that would be missing are the data values and the constraints.

Even though a component may have many internal models of different types, it may be conceived as having a single interface (external model)– albeit at many different levels of abstraction and possibly along different axes, as defined in this taxonomy. As an example of this approach, rather than stating "this is an interface model for the behavioral-level model of component X," we state "this interface model for component X contains the behavioral level."

The rationale for such a multi-level interface model is that an interface specification at a lower level of abstraction for a component must conform to all interface specifications at higher levels of abstraction along the same axis of description. The lower level of abstraction may have more detail, but it should not go beyond the design space delimited by the higher level specifications. As all the levels must be consistent with each other in this way, it is useful to provide the different levels in a single interface model.

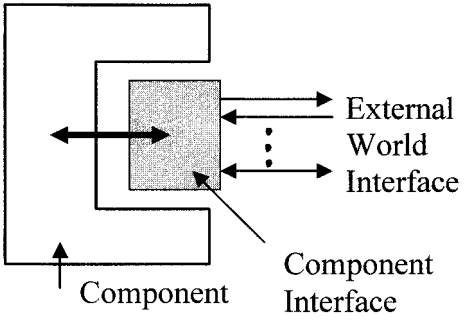
2.2.1.5 System-Level Interfaces

The possibly multi-level nature of interfaces, as described in the last section, can lead to interesting juxtapositions of models. For example, it is possible to have a component model at one level of abstraction that is actually used by the system at a different level of abstraction, with the translation between the levels being provided by the interface model. This

means the *internal* level of abstraction of a component can be different from the *external* level of abstraction that is used by the system via that component's interface.

Since all components must have some sort of interface model at some level of abstraction, even if it is a rudimentary model, we can make some useful definitions while specifying requirements on components and their interfaces at system level.

So, as in Figure 2.17, an interface model would sit between the internal model and the environment.



The levels of abstraction available to the external world from a component interface may be different to (or a superset of) the level of abstraction of the component itself.

Figure 2.17 System-Level Interfaces

Components for which there may, or may not, be a realization are termed *virtual components* (VCs) and their interfaces termed *virtual component interfaces* (VCIs). All components are defined with an interface to the external world. See Section 2.9.9.1, "General Interface Terms." In all places where components are mentioned, the references can equally apply to virtual components, and vice versa.

This particular interface is responsible for the transfer of information from the internal abstractions of the component to information in a form compatible with the abstractions in the external world to which the component interfaces. The models of these worlds might be at quite different levels of abstractions along, say, the protocol and data axes. For example, the internal world of a component may be at a "general system level" using named values and loose timing whereas the external world interface may require signal and port specifications and behaviors with additional timing detail and at bit-accurate level.

The block that performs the translation from the internal world to the external world is the component interface. The interface translation may come in several differing flavors, depending upon the application (for

example, hardware component-to-peripheral bus, hardware component-to-system bus, and software component-to-system bus).

The multi-level definition of the component interface permits the nesting of interface abstractions, as depicted in Figure 2.18.

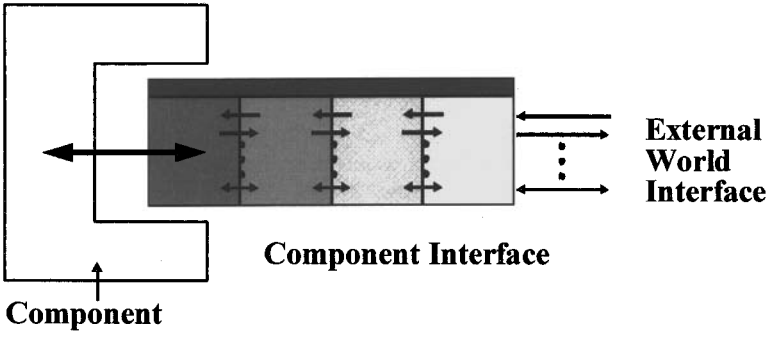


Figure 2.18 Nesting of Interface Abstractions

This nesting of abstractions manifests itself as a hierarchy of interface layers. Depending upon the specific component type and interface, these layers may be fully specified in terms of predefined properties. In other cases, arbitrary interface hierarchy may be permitted as long as specifications for relating the levels of the abstraction hierarchy is given

2.2.2 Specialized Model Classes

The following model classes describe models intended for specific purposes that are not unique to a particular level of abstraction.

2.2.2.1 Performance Model

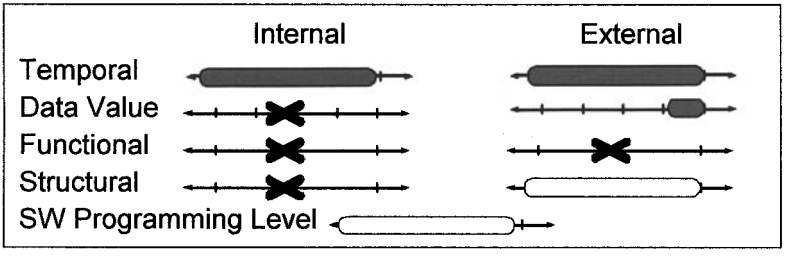


Figure 2.19 Performance Model

Performance is a collection of the measures of quality of a design that relate to the timeliness of the system in reacting to stimuli. Measures associated with performance include response time, throughput, and utilization. A performance model may be written at any level of abstraction. In general, a performance model may describe the time required to perform simple tasks such as memory access of a single CPU. However, in the context of component reuse, the typical abstraction level for performance models is most often at the multiprocessor network level. For clarity, such a model is called a *token-based performance model.* (See Section 2.4.1.)

Although performance models here are described solely in terms of design timing or delay, there are other attributes that are often lumped together with timing attributes as aspects of "performance models." These can include, for example, power consumption, design cost, reliability, maintainability, and other system-level attributes. In general, almost any measurable quantity can give rise to a performance model for a component. Although Section 2.7 discusses a number of "implementation-level performance models" that include power models, this more general use of the term "performance" is not universal, and can cause considerable confusion if used generically without qualification. Therefore, we recommend that the unqualified term "performance" generally be taken to refer to timing or delay characteristics; and that use of "performance" to represent power, cost, and so on, always be qualified in some fashion. Of course, it is always best to make the use of any term unambiguous.

2.2.2.2 Mixed-Level Model

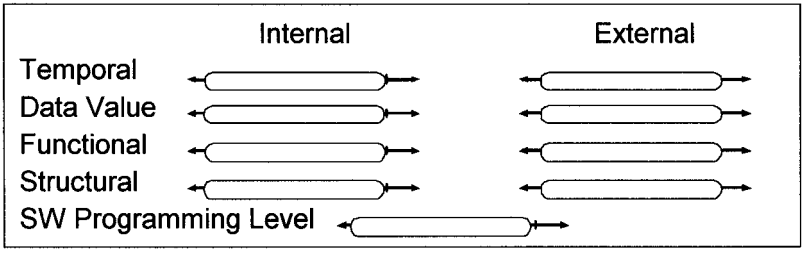


Figure 2.20 Mixed-Level Model

A *mixed-level* model is a combination of models of differing abstraction levels or descriptive paradigms. Such a model is sometimes called a *hybrid* model. However, this taxonomy prefers the term "mixed-level" over the term "hybrid" because the former is more specific.

2.2.3 Computation Model Classes

In this section, a number of models that are computational in nature will be introduced. Computation models can be the underlying engine for several different abstraction models. It should be noted that different classes of computation models are often referred to in the literature and common use as "models of computation."

2.2.3.1 Dataflow Graph Model

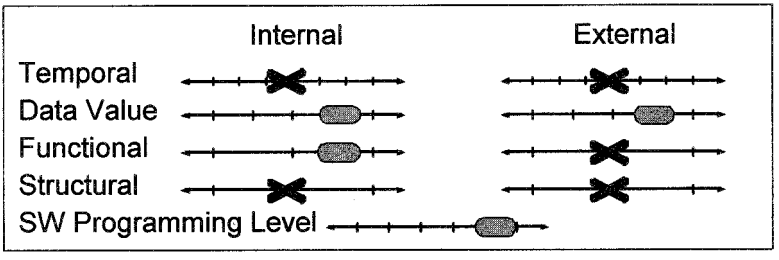


Figure 2.21 Dataflow Graph Model

A *dataflow graph* (DFG) model describes an application algorithm in terms of the inherent data dependencies of its mathematical operations. The DFG is a directed graph containing nodes that represent mathematical transformations and arcs that span between nodes and represent their data dependencies and queues. The graph conveys the potential concurrencies within an algorithm, which facilitates parallelization and a mapping to arbitrary architectures. The DFG is an architecture-independent description of the algorithm. It does not presume or preclude potential concurrency or parallelization strategies. The DFG can be a formal notation that supports analytical methods for decomposition, aggregation, analysis, and transformation. The DFG nodes usually correspond to DSP primitives such as FFT, vector multiply, convolve, or correlate. The DFG graph can be executed by itself in a data-value-true mode without being mapped to a specific architecture, though it cannot resolve temporal details without co-simulation with an architecture-performance model.

The primary purposes of a dataflow graph are to express algorithms in a form that allows convenient parallelization and to study and select optimal parallelization or execution strategies through various methods involving the aggregation, decomposition, mapping, and scheduling of tasks onto processor elements.

A DFG is a directed graph with actors at the nodes. The arcs represent the flow of data, and behave conceptually as FIFO queues. DFGs are characterized by the fact that they do not over-specify an algorithm by unnecessary sequencing constraints between the operators in the graph. DFGs specify partial, relative orders by indicating only data dependencies. Such a model is well suited to exploit maximal parallelism and to reuse or share different hardware resources or processing units in time. Therefore, compilers for partitioning on pipelined or parallel processors often rely heavily on dataflow analysis.

Synchronous dataflow graphs are special cases of DFGs, because the number of tokens consumed or produced by a given actor when it fires is fixed and known at compile time. In order to express data-dependent iteration, conditionals, and recursion, a more general DFG model is needed (dynamic dataflow).

2.2.3.2 Other Computational Models

Other potential candidates for computational models include the following: Petri nets, discrete events, process networks, reactive models, communicating sequential processes, and hierarchical communicating FSMs.

We will give a brief summary of models of computation based on the taxonomy developed and published by Axel Jantsch [JANTSCH].

Jantsch defines a "model of computation" or "model of concurrency" (MOC) as those modeling aspects that are relevant for the communication, synchronization, and relative timing of concurrent processes [JANTSCH, p. 4]. His taxonomy of systems and models includes important system properties such as:

- State-less and state-full systems
- Time-varying and time-invariant systems
- System-state: continuous state and time; discrete state and time.
- Linear and nonlinear systems
- Deterministic, stochastic and nondeterministic systems
- Events
- Time-driven and event-driven systems

He uses these properties to define a set of different system models and a taxonomy to classify them. For example, he suggests that in control theory, state-full, time-invariant, linear, continuous-state, continuous-time systems are studied. In electrical engineering, analog designers work with nonlinear, continuous-time, continuous-state systems. Software engineers work with discrete-state, discrete-time systems.

His taxonomy then concentrates mostly on state-full, time-invariant, nonlinear, discrete-state, discrete-time systems. He then presents a meta-model, called "Rugby," that links hierarchy, abstraction, and domains (time, data, computation and communications) together.

The behavioral models discussed by Jantsch fall primarily into two categories – finite state machines (FSMs), and Petri nets. FSMs come in a multitude of types: nondeterministic, nondeterministic with epsilon-moves, FSMs with outputs (Moore machines, where output events are emitted with a state, and Mealy machines, where output events are emitted with a transition), and FSM extensions including push-down automata, FSM with datapath, Harel statecharts, and co-design finite state machines.

Petri nets are introduced as a formalism that allows certain analyses not possible with FSMs. Indeed, if a design problem can be formulated as a Petri net, a number of powerful results can be obtained using formal techniques.

The next MOC discussed by Jantsch is the untimed MOC. This is based on notions of processes and signals, where signals consist of sequences of events. Jantsch uses event sequences including absent events for timed event sequences, in contrast to the tagged signal model of Lee and Sangiovanni-Vincentelli [LEE 98]. Processes are related to functions on events via process constructors which are higher-order functions that return processes. The constructors include maps (no internal state), scan (internal state and a next-state function), Moore (output is function of state) and Mealy (output is function of state and current input). A series of properties and analysis then yields a formal mathematical definition of the untimed MOC, and a set of techniques to allow untimed MOC models to be combined and analyzed. In a rigorous fashion, dataflow models such as synchronous dataflow (SDF), and variants such as Boolean and cyclo-static dataflow can then be built on top of the untimed MOC.

The Synchronous MOC divides time into slots and come in two flavors in Jantsch's analysis: perfectly synchronous and clocked synchronous. Synchronous MOCs are the basis for several research languages in the control-oriented domain (StateCharts, Esterel, Argos) and dataflow-oriented domain (Signal and Lustre).

The timed MOC, which again is based on the notions of process constructors, is a generalization of the synchronous MOC, and encompasses discrete event models using delta-delays, which are the basis for most discrete event simulators such as HDL simulators (Verilog, VHDL, SystemVerilog) and other examples such as SystemC 2.0.

Important in Jantsch's MOC framework is a set of formalisms allowing interfaces between MOCs to be constructed and analyzed. This includes different domains of the same MOC, and interfaces between different computational models. This is based on a set of interface processes (insertion

and stripping) which act between the timed, synchronous and untimed MOCs to allow the filtering out, or insertion of, appropriate timing information, such as to permit the composition of models to still produce meaningful results. This is not yet susceptible to a systematic and rigorous formalism but is based on a number of practical solutions. Practical approaches using the notion of interfacing MOCs include Ptolemy, composite signal flow and trace algebra-based approaches.

Jantsch concludes his MOC treatment with a consideration of tightly-coupled process networks, and nondeterminism in dataflow networks and process algebras such as Hoare's Communicating Sequential Processes and Milner's Calculus of Communicating Systems.

Although Jantsch's framework is not complete, in that it does not encompass every MOC, it is quite comprehensive and applies a formal mathematical rigor to the issues of MOC definition, analysis and interaction. It is interesting to note that such models in general cannot fit into the taxonomy axes as discussed previously except in the most general way (such as the interface notion of time) because in general "events" and the signals that they build up as interfaces can be at multiple levels of abstraction.

2.3 Other System Models

The following sections define terms used to abstractly describe models of digital electronic systems, such as digital signal processing (DSP) systems or control systems. The abstraction does not include any information about any hardware or software structure for implementing the system. The models in Section 2.2, "General Modeling Concepts" can also be implementation-neutral, and hence are classified as possible "system models."

2.3.1 Executable Specification

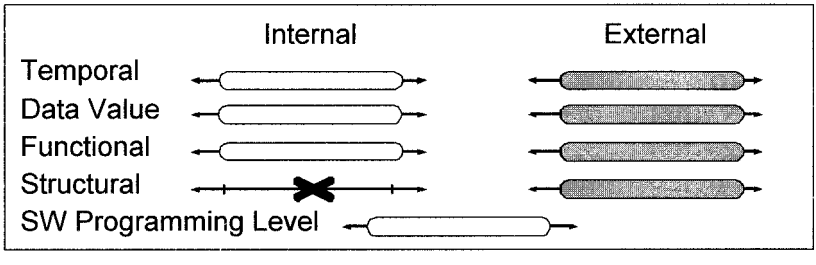


Figure 2.22 Executable Specification

A specification has traditionally been a static description of a component and its characteristics, in multiple dimensions, on paper or an electronic equivalent (for example, a data sheet in paper or electronic form). Specifications in the past have not been executable, but the electronics design world is beginning to move to an executable format. The introduction of modeling languages, such as SystemC, has enabled the industry to create higher-level models that are interchangeable and that support the notions of the separation of function and interface.

An *executable specification* is a behavioral description of a component or system object that reflects the particular function and timing of the intended design as seen from the object's interface when executed in a computer simulation. The executable specification may also describe the electrical or physical aspects including the power, cost, size, fit, and weight of the intended design. Denotational items such as power are normally considered factual (derived) items to be checked but not executed. Executable specifications describe the behavior of an object at the highest level of abstraction that still provides the proper data transformations. (Correct data in yields correct data out; *defined* bad data in has the *specified* output results.) Executable specifications may describe an object at an arbitrary abstraction level such as a multiprocessor system, architecture, or hardware or software component level.

The primary purpose of an executable specification is for testing that the specified behavior of a design entity satisfies the system requirements when integrated with other components of the system, and for testing whether an implementation of the entity is consistent with the specified behavior.

2.3.2 Mathematical-Equation Model

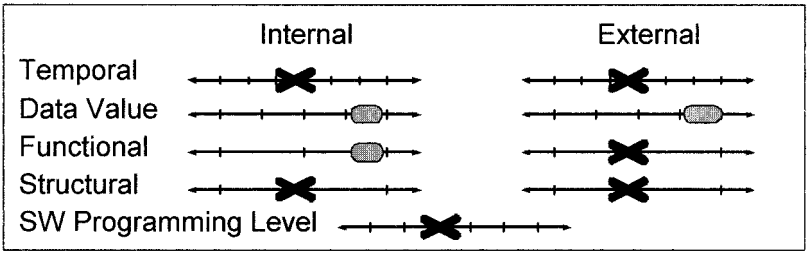


Figure 2.23 Mathematical-Equation Model

The mathematical-equation model describes the functional relationship of input to output data values. A mathematical-equation model is a purely algebraic expression of the function the target system is to provide. The

mathematical model is differentiated from an algorithm description in that a mathematical model does not imply a specific sequence of operations to implement the function. Examples of mathematical descriptions for system functions include:

$y = sqrt(x)$

and

$y = sin(x)$

These functions represent well defined mathematical relationships, but do not indicate methods for their computation, for which there are many, such as look-up table, Newton's method, or Taylor-series expansion.

The primary purpose of a mathematical-equation model is to test that the mathematical equations and parameters developed to solve a design challenge do satisfy a system's numerical performance requirements.

2.3.3 Algorithm Model

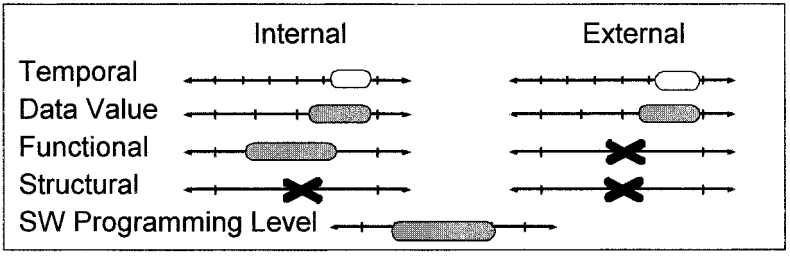


Figure 2.24 Algorithm Model

The algorithm level of abstraction describes a procedure for implementing a function as a specific sequence of arithmetic operations, conditions, and loops. An algorithmic description is less abstract than a purely mathematical description because it provides more detailed information for implementing the function(s). An algorithm model transforms actual data. Examples of algorithms include quick-sort, Givens triangularization, Cholesky matrix decomposition, bisection method, Cooley-Tukey FFT, and Winograd FFT.

The primary purpose of an algorithm model is to test how well an algorithm designed to implement a mathematical task satisfies the system numerical performance requirements. Algorithm models are also used for determining the numerical effects of finite precision and the parameters of floating or fixed formats.

In the analog/mixed-signal community this is called an *algorithmic-level model.*

2.4 Architecture Models

The following sections define terms used to describe some abstract models of a system's hardware and software architecture. As such, these models describe only the basic structure of the application and the hardware to which the structure can be mapped. Details that are not relevant to the architecture design process are relegated to the detailed hardware and software models.

2.4.1 Token-Based Performance Model

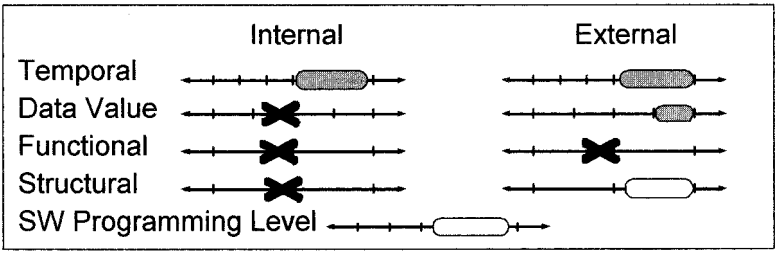


Figure 2.25 Token-Based Performance Model

The *token-based performance* model is a performance model of a system's architecture. (See Section 2.2.2.1, "Performance Model.") Measures associated with performance include response time, throughput of the system, and the utilization of the system resources. Typically, the token-based performance model resolves the time for a system to perform major system functions. Data values are not modeled, except for control information. The structure of the system is described down to at least the major node level. The internal structure of the switches, processor elements, shared memories, and I/O units are not usually described in a token-based performance model. The primary purpose of a token-based performance model is to determine the sufficiency of the following system properties in meeting the system processing throughput and latency requirements: the number and type of elements, the size of memories and buffers, the network topology (bus, ring, mesh, cube, tree, or custom configuration), network bandwidths and protocols, application partitioning, mapping, scheduling of tasks onto processor elements, and flow control scheme.

Token-based models may also be used at functional or behavioral levels (for example, to investigate communication densities), although their most common use is at the architectural level in evaluating different architectural patterns.

2.4.2 Abstract-Behavioral Model

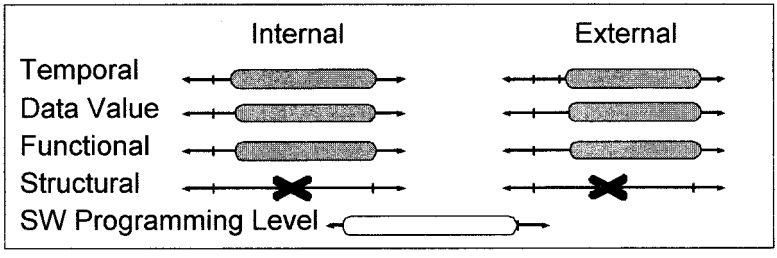


Figure 2.26 Abstract-Behavioral Model

Abstract-behavioral models encompass a wide range of modeling abstraction levels on the temporal, data and functional axes–from composite-passing abstract performance levels to detailed data-accurate and fine-grained temporal and functional models. The "detailed behavioral model" described in Section 2.5.1 differs from the lower abstraction end of this model primarily in that its interface is usually hardware-component specific.

An abstract-behavioral model can model a system application on which the performance effects of the particular system architecture chosen to implement that application have been annotated using abstract performance models.

This behavioral model is expressed at a token-passing level, but data values within the tokens are both modeled *and* functionally processed (in contrast to the token-based performance model). In other words, the system application behavior is accurately modeled and the tokens represent real data passing between system functions as well as control messages.

An abstract-behavioral model's interface is modeled abstractly. The model does not resolve the interface ports to their pin structure. For instance, a microprocessor's memory interface may be described as a single port having a complex data type, as opposed to specifying the constituent control lines, and address and data buses and their bit-widths. See Section 2.2.1.2, "Behavioral Model."

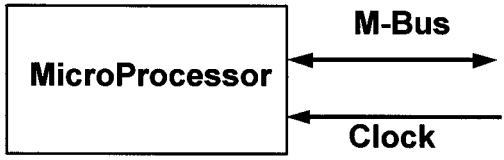


Figure 2.27 Memory Interface

The abstract-behavioral models are used to annotate performance effects onto the system-application functional blocks and inter-block communications mechanisms. Thus, they model estimates of functional processing time for blocks, inter-block communications protocol delays, contention for shared resources such as processors (microprocessors, controllers and DSPs), and shared communications resources such as buses and associated memories. The internal structures of processors, dedicated hardware blocks, and detailed wiring-level models for buses are not usually described in this level of modeling. However, communications protocols between blocks may be described at both a high-level, complete token transfer, and at a finer level of granularity, in which tokens are broken up into a sequence of generic communications and bus transactions.

The usual primary purposes of a behavioral, composite-passing, abstract-performance model are to determine the suitability of an architecture as a base on which to map a set of intended application behaviors. Thus the number and power of processors, dedicated hardware units, communications buses and protocols, sizes of memories and their number and distribution are all questions that may be answered at this "architectural trade-off" level. In addition, the evaluation of abstract models of third-party IP blocks may be carried out using this kind of model, to determine what best meets system requirements.

Other purposes for this model type are:

- To establish and verify the joint functional and timing requirements for the components, to ensure that collectively they are consistent with the overall system requirements
- To verify the numerical correctness of the hardware/software mapping as modeled by the performance model
- To facilitate reuse of the system design when implementation is changed for the components or interfaces
- To produce test-vectors for use in the detailed design of the components and to aid in system integration, diagnosis, and testing
- To help visualize the operation of complex systems for understanding its characteristics for optimizing the design, especially at the board level prior to making detailed decisions about

the exact nature of component interfaces–to accomplish top-down design
- To document the intended operation of the system implementation

This kind of model is common where design above RTL is practiced and is usually ambiguously called a "high-level model." However, the term *abstract-behavioral model* should now be used. This is sometimes called a *system evaluation model* or *behavioral model* when additions of analog-performance characteristics are made in addition to timing.

2.4.3 Dataflow Graph (DFG) Task Primitive

Figure 2.28 Dataflow Graph (DFG) Task Primitive

Software at the task-primitive level expresses the application in terms of its building-block functions. These functions may ultimately be implemented in hardware or software. They are usually expressed in a graphical form.

2.4.3.1 Instruction-Set Architecture (ISA) Model

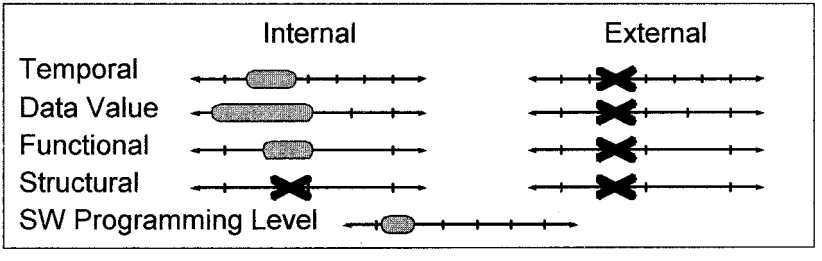


Figure 2.29 Instruction-Set Architecture (ISA) Model

An ISA model describes the function of the complete instruction set recognized by a given programmable processor, along with (and operating on) the processor's externally known register set and memory or input-output (I/O) space. An ISA model of a processor will execute any machine program for that processor and give the same results as the physical machine, as long as the initial states (and simulated I/O) are the same on the ISA model simulation as they are on the real processor. Such a processor model with no external ports is classed as an ISA model. If the processor

model has external I/O ports, then it would be classified as a behavioral model.

Data transformations of ISA models are bit-true, in terms of word length and bit values as observable in the internal registers and memory states. Port buffer registers, if modeled, are also bit-true. The temporal resolution of an ISA model is at the instruction cycle. Instruction cycles may span multiple clock cycles. An ISA model contains no, or relatively little, internal structural implementation information. It may contain enough details about the processor internal pipeline and other internal structures in order that it can give correct instruction-accurate level results. It is also possible that ISA models can be written to be cycle accurate and yet still perform with sufficient speed for SW development and debugging. At the time of publication, it is clear that ISA models, otherwise known as instruction-set simulators (ISSs), exist at both instruction-accurate and cycle-accurate levels. Sometimes this is accomplished with the same core model and different external interface wrappers.

The primary purposes of ISA models are for efficient development of uniprocessor resident software prior to hardware realization, optimization and design of application-specific instruction sets and register architectures, documenting the functionality of the processor's instruction set, and measuring software routine run-times, to increase the accuracy of the more abstract models.

2.5 Hardware Models

This section defines terms used to describe the various types of hardware models. These models are used to describe the hardware at specific levels of abstraction.

2.5.1 Detailed Behavioral Model

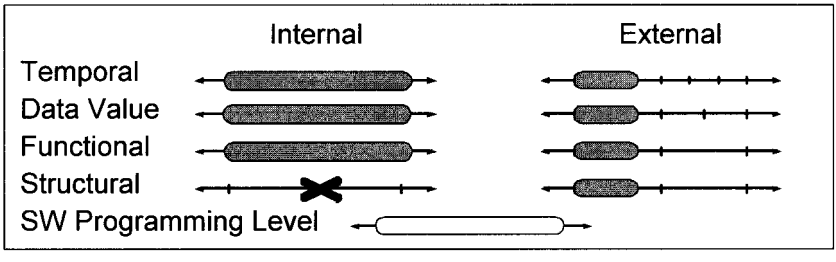


Figure 2.30 Detailed Behavioral Model

The *detailed behavioral* model is a behavioral model that describes the component's interface explicitly at the pin level. It exhibits all the documented timing and functionality of the modeled component, without specifying internal implementation structure. This type of model has traditionally been called a "full-functional model" and is therefore a synonym. However, the newer term is preferred for its better accuracy and consistency to the definitions of the related models.

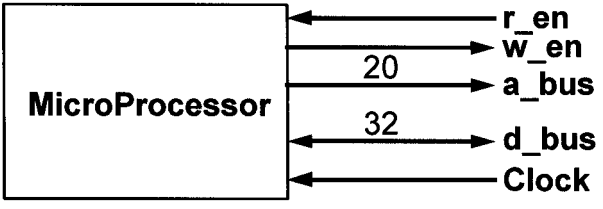


Figure 2.31 Detailed Behavioral Model Interface

The primary purpose of a detailed behavioral model is to develop and comprehensively test the structure, timing, and function of component interfaces. It also helps examine the detailed interactions between hardware and software (drivers), and provides timing values that are used to replace initial estimates in the higher-level models to increase their accuracy. In the analog/mixed signal world, this is called a "functional/timing digital simulation model."

In the system TLM world this is the model that corresponds to the cycle-callable (CC) model, alternatively called the "bus–cycle-accurate transfer layer (BCA-TLM)." It provides an accurate view of the interface of a block without placing any constraints on the internal implementation. This makes it useful for the verification of the detailed interactions between hardware and software systems.

2.5.2 Register-Transfer-Level (RTL) Model

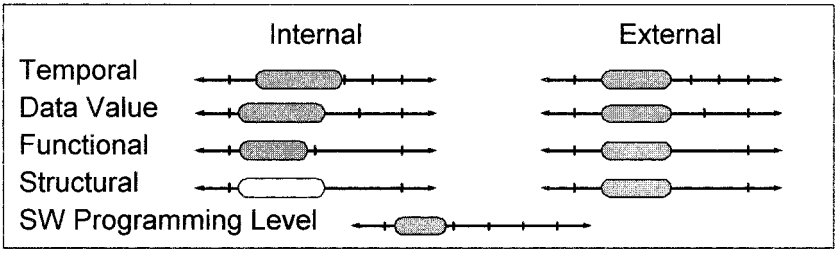


Figure 2.32 Register-Transfer-Level (RTL) Model

An *RTL* model describes a system in terms of registers, combinational circuitry, low-level buses, and control circuits, usually implemented as FSMs. Some internal structural implementation information is implied by the register transformations, but this information is not explicitly described.

The primary purpose of RTL models is for developing and testing the internal architecture and control logic within an IC component so that the design satisfies the required functionality and timing constraints of the IC. The RTL model is also used for specifying the design in a process-neutral format that can be targeted to specific technologies or process lines through automatic synthesis. It is often used for generating detailed test vectors, gathering timing measurements to increase the accuracy of more abstract models, and investigating interactions with closely connected components. It unambiguously documents the design solution.

2.5.3 Logic-Level Model

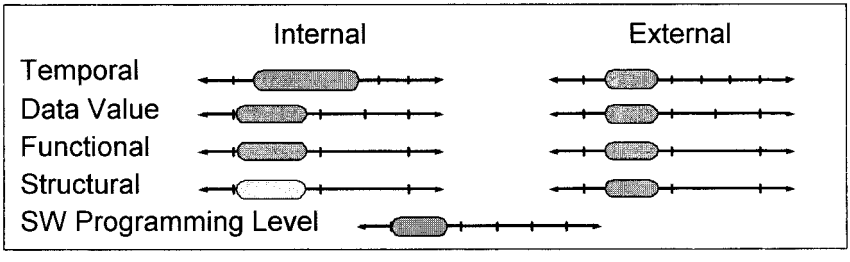


Figure 2.33 Logic-Level Model

A *logic-level* model describes a component in terms of equivalent Boolean logic functions and simple memory devices such as flip-flops. The logic-level model does not describe the exact implementation in logic gates. The logic expressions can be transformed or reduced into functionally equivalent forms prior to target implementation in logic blocks.

The primary purpose of logic models is to develop logical expressions for reduction into logic gates, and to test that these expressions implement the required functionality, usually for a portion of an IC component. They also support re-use of a detailed design by documenting the logic in a fairly process-neutral format that can be targeted to specific technologies or process lines through automatic synthesis.

2.5.4 Cell-Level Model

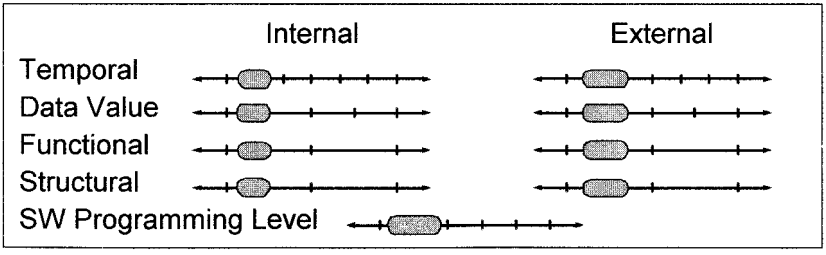


Figure 2.34 Cell-Level Model

A *cell-level* model describes the function, timing, and structure of a component in terms of the structural interconnection of Boolean logic blocks. The Boolean logic behavior blocks implement simple Boolean functions such as NAND, NOR, NOT, AND, OR, and XOR. A cell-level model describes the actual structure and versions of cells that are assembled to implement the target component.

The primary purpose of a cell-level model is to document the particular implementation of an IC component in terms of the interconnection of elements from a specific logic-family library, for fault-grading and production of operational test-vectors, to determine the precise timing response of the circuit to stimuli to increase the accuracy of more abstract models, to test that the design meets all timing and functionality requirements, and to optimize the cell-level implementation of the logical design.

Since these logic-family libraries are most often digital standard cell libraries, or the equivalent for gate arrays, structured ASICs or programmable logic, we refer to this model as a cell-level model rather than the more common, but less precise, "gate-level model." Often, a standard cell maps into a complex Boolean function or a combination of several functions (such as AND-OR-INVERT).

2.6 Switch-Level Model

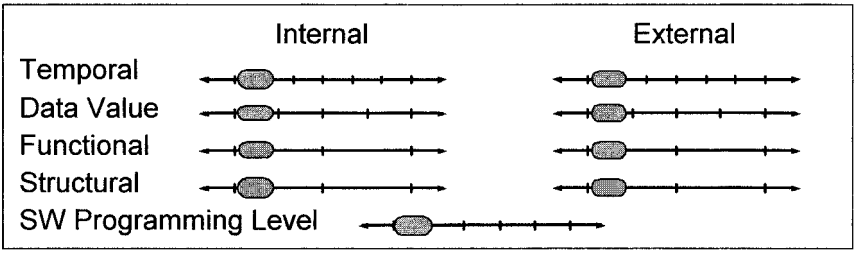


Figure 2.35 Switch-Level Model

A *switch-level* model describes the interconnection of the transistors composing a logic circuit. The transistors are modeled as simple, voltage-controlled, on-off switches.

The primary purpose of a switch-level model is to efficiently determine the response of a portion of an IC, usually a gate or set of gates, to increase the accuracy of more abstract models. This determination is more efficient though coarser than circuit-level models.

2.6.1 Circuit-Level Model

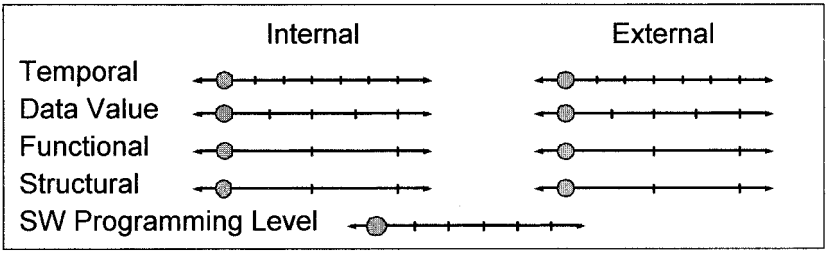


Figure 2.36 Circuit-Level Model

A *circuit-level* model describes the operation of a circuit in terms of the voltage-current behaviors of resistor, capacitor, inductor, and semiconductor circuit components and their interconnection.

The primary purpose of a circuit model is to determine the response of a portion of an IC, usually a gate or set of gates, to optimize its design according to its requirements. It can accurately determine the design's minimum and maximum propagation, switching times, and loading and driving capabilities in terms of transistor and conductor properties and

configurations, which increase the accuracy of more abstract switch-and gate-level models.

2.7 Implementation-Level Performance Models

These models define data required for timing and power-driven implementation and verification of an SoC design incorporating Components. The models complement many of the models in Section 2.5, "Hardware Models," and support design phases of logic synthesis, static timing analysis, and power analysis.

2.7.1 Basic Delay Model

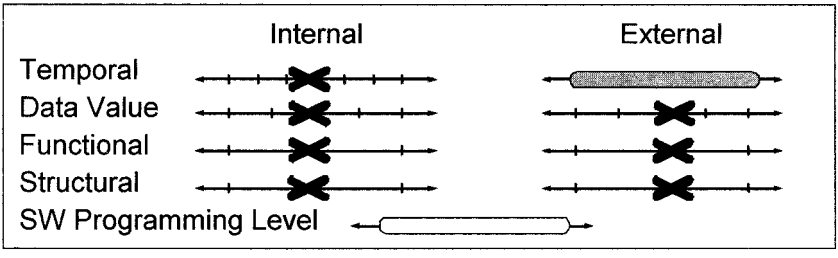


Figure 2.37 Basic Delay Model

The *basic delay* model defines the timing specification of the component. It is required for delay calculation, and is the basis for the "timing analysis model" described in Section 2.7.2. The basic delay model includes path delays associated with timing arcs; signal slew rates for output signals; and timing checks, associated with timing arcs. The delay calculation model should include the dependence of signal delays and slews on the environment of the component, given an appropriate implementation technology.

2.7.2 Timing Analysis Model

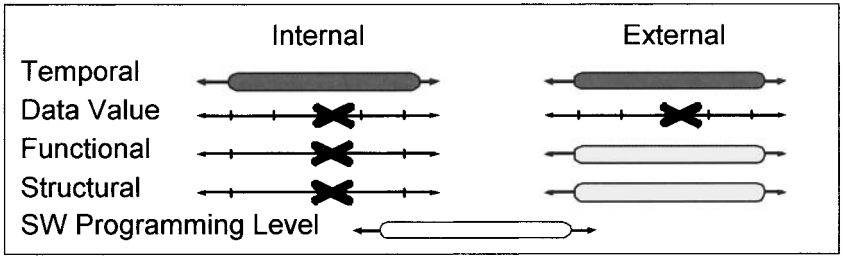


Figure 2.38 Timing Analysis Model

The timing analysis model describes the static timing characteristics of the component, including interface and timing arc attributes that are not included in the "basic delay model" described in Section 2.7.1, but are necessary for static-timing analysis. The model includes state-dependent timing and modes of operation; insertion delay and skew of clock networks within the block; multiple operating conditions; design properties; multi-cycle and false paths; the parasitics of peripheral interconnect (for use at circuit level), and physical connection points (for use at circuit level).

2.7.3 Power Model

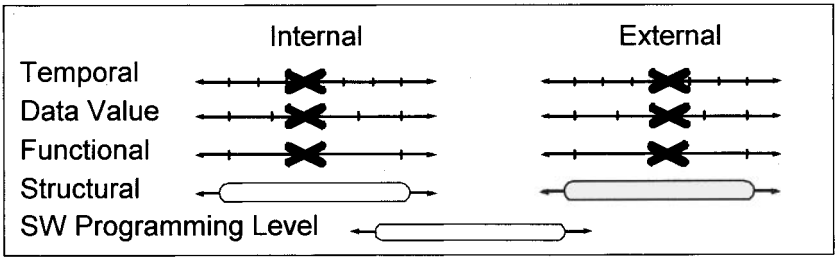


Figure 2.39 Power Model

The power model defines the power specification of a component. This may depend on one or many atomic power models, which contain no hierarchy and no dependencies on other power models. A power model should be able to represent both dynamic and static power. Power models may be provided with various levels of transparency and accuracy, such as black-box or gray-box requirements, RTL source, cell-level netlist, and circuit-level (transistor) netlist.

Note that the axis system described earlier proves less useful in describing power models, since there is nowhere to indicate the power values that such models generate. This would generally be true of many other attributes of a system such as size, cost, reliability, and so on. However, there is in most cases a connection between the ability to estimate these attributes and the resolution of the model used to predict them.

2.7.4 Peripheral-Interconnect Model

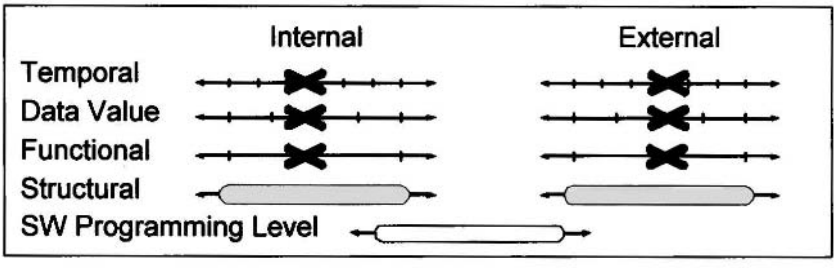


Figure 2.40 Peripheral-Interconnect Model

A *peripheral-interconnect* model represents an interface interconnection network "shell" around an internal component-delay model. It specifies interconnect RCs (resistances and capacitances) between physical I/O ports and the internal gates of a component. This provides a separation of peripheral interconnect RCs from the component intrinsic delays. Thus, delay calculation at the next level can be performed using actual interconnect rather than an inaccurate approximation of loading and interconnect.

Note that the axis system described earlier proves less useful in describing peripheral-interconnect models, since there is nowhere to indicate the RC values such models generate.

2.8 Software Models

The following sections define the forms of software and the levels of abstraction in the software hierarchy. For a more detailed discussion of the concepts associated with software and the connection between the hardware and software components of a system, the reader should also refer to Chapter 5, "Hardware-dependent Software."

2.8.1 Requirements Modeling

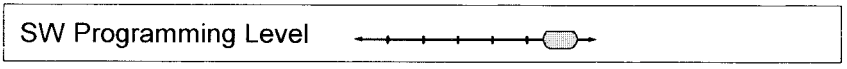


Figure 2.41 Requirements Modeling

Requirements or *specification* modeling is done at the very high level for a system or for software. Examples of models at this level are those generated using object-oriented analysis in schemes such as OMT (Object Modeling Technique) and UML (the Unified Modeling Language).

2.8.2 Pseudo-Code

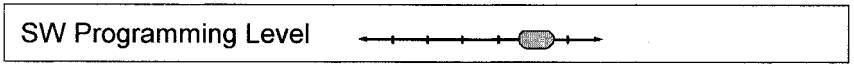


Figure 2.42 Pseudo-Code

Pseudo-code is a simplified or abstracted code form that is used as an intermediate step toward preparation of standard high-level-language code. Some UML models, such as state diagrams annotated with the Action Semantics language, can be thought of as pseudo-code, and tools may be able to use this code to generate executable code in standard languages such as C, C++, or Java.

2.8.3 High-Level Language (HLL)

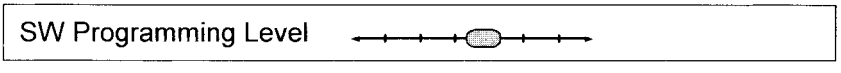


Figure 2.43 High-Level Language

High-level-language software is a machine-independent (retargetable) form of software that conforms to a standard language grammar and syntax. It is characterized by text-based arbitrary symbolic variable names and control constructs, and uses algebraic expression statements. Examples include many languages in use over the years, including FORTRAN, COBOL, C, Ada, C++, Java, and others.

2.8.4 Assembly Code

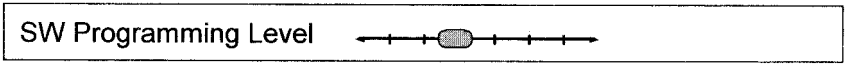


Figure 2.44 Assembly Code

Assembly code is mnemonic-based object code. For more information, see Section 2.8.6, "Object Code." Assembly code tends to be text-based, but with limited expression syntax, usually restricted to a set of explicit operators and register names. Operators tend to be simple arithmetic and logic operations. Each line typically specifies one operation per instruction cycle. Data variables are usually related to specific memory addresses. Very Long Instruction Word (VLIW) machines or multi-issue (multi-operation) machines may have assembly formats in which multiple operations that are executed in one instruction cycle are specified on one line of assembly, separated by standard delimiters (such as ";"). This is not to be confused with microcode, as discussed in the next section, in which the control instructions control the various portions of the machine micro-architecture at a fine-grained level; multi-operation assembly instructions are at a higher abstraction level.

2.8.5 Microcode

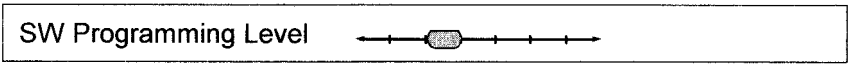


Figure 2.45 Microcode

Microcode consists of machine-executable control instructions that individually control phases of execution and internal processor structures. It is much like assembly code, but instead of specifying one operation per instruction cycle as in assembly code, microcode specifies the settings for several units, such as buses, multiplexers, and function units for each clock cycle to accomplish the type of operation typically specified by a single assembly instruction. Several microcode instructions typically compose a traditional assembly instruction in a microcode-based architecture. Each assembly-level instruction is typically decomposed into a microcode routine which is a set of microcode instructions. See Section 2.9.2.7, "Firmware."

Microcoded machines have gone into and out of fashion over the past years, but should not be confused in general with multi-operation, multi-issue machines such as VLIW processors.

2.8.6 Object Code

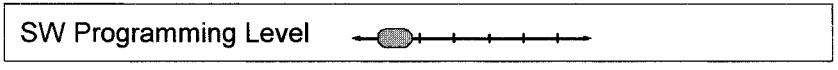


Figure 2.46 Object Code

Object code consists of machine-executable control instructions. There is usually no symbolic representation in object code. The code is expressed directly in the form acceptable by the digital logic on the processor, usually as numeric ones and zeros. Object code is composed of the strings of ones and zeros that are used directly by the hardware, whether it is coming from ROM, RAM, or direct input. HLL compilers, assemblers at the assembly level and microcode-generation systems all produce object code. Object code is also called *machine code*.

2.9 Supporting Terms

The following sections list a collection of terms and their definitions that support the definitions in the previous sections. These terms tend to be more general than the previous model-specific terms, yet clear interpretations of these terms are not often found within our context. The previous definitions rely on an unambiguous understanding of their meanings.

2.9.1 Abstraction Level and Hierarchy

2.9.1.1 Abstraction Level

The *abstraction level* is an indication of the degree of detail specified about how a function is to be implemented. Abstraction is inversely related to the resolution of detail. If there is much detail, or high resolution, the abstraction is said to be low. More implementation details become constrained as the abstraction level is lowered.

The abstraction levels form a hierarchy. A design at a given abstraction level is described in terms of a set of constituent items and their inter-relationships, which in turn can be decomposed into their constituent parts at a lower level of abstraction.

For example, consider the function:

c = a convolved with b

This function is more abstract than the following:

*c = idft(dft(a) * dft(b));*

The second function shows more information about how to compute (or implement) the function. The function could have been implemented in other ways (such as $c = S\ ai{+}k * bn{-}k$), so the second equation provides details that are not contained in first equation.

An even lower level of abstraction for the example would be an implementation of the equation in the form of a multiplier or accumulator with an FSM controller. This description provides more constraining information about the implementation, since the second equation could have been implemented and described in other ways, such as a software-programmed computer.

A still lower abstraction level would be a logic-gate netlist for the FSM and multiplier or accumulator. For example, it would resolve more details about how to implement the adder function, and therefore it would constrain the implementation further.

An even lower abstraction level would be the polygon layout for the logic, since more details would be resolved and constrained.

It should be evident that many intermediate levels of abstraction were skipped in this example.

It should be noted that abstraction level does not indicate accuracy. Abstraction and accuracy should be distinguished in the same sense that precision is different from accuracy. For example, an abstract-behavioral model of a deterministic processing element could describe the execution of a given function as consuming 100.288 microseconds, while a much less abstract RTL model would concur with exactly the time in terms of 100288 1-nanosecond (1 GHz) clock cycles. Thus, both models can be equally accurate but differ in abstraction.

2.9.1.2 Hierarchy

Hierarchy is a multi-level classification system that supports aggregation and decomposition. A node at a given level of the hierarchy can be represented by the set of its descendant nodes (and their inter-relationships) on the next lower level.

The important hierarchies include the functional or logical hierarchy and the physical hierarchy. Often there is a correspondence between these hierarchies, but not always.

A functional hierarchy decomposes a system according to its functional parts, such as receiver, product detector, convolver, multiplier or accumulator, register or multiplexer, and logic gate.

A physical hierarchy may decompose a system according to its physical structure, such as racks, frames, chassis, boards, modules, chips, cells, gates, and transistors.

The functional–to-physical-structure mapping tends to shift with model year as integration levels increase.

2.9.2 Design Object Classes

2.9.2.1 System

A *system* is any composition of parts that performs a function or set of functions. The boundaries of a system usually follow the structural implementation, but may also cross physical boundaries. For instance, a memory system XYZ might share boards P, Q, R, and S with other systems. Systems are typically hierarchical in that a system may be composed of sub-system components. A system is characterized by the interrelations and behaviors of its components.

Examples of systems are:

1. Digital signal processing system, which is composed of sub-systems such as:
    - I/O system
    - Network communication system
    - Local and distributed operating systems
    - Run-time command processing and control systems
    - Processor clusters, or multi-node board systems
    - Power supply system
2. Multi-node, processor-board system, composed of sub-systems such as:
    - Processor-element subsystems
    - Shared memory system
    - Scan-chain control system
3. Processing element system (analogous to a CPU system), composed of sub-systems such as:

- Local memory system
- Local node operating system
- Processor unit sub-system
- Inter-PE communication system
- Run-time, node-control system

To make the term system clear, it is recommended that the appropriate qualifier be stated before its use, such as "CPU memory system," "radar system," "DSP system," and so on.

Most systems are components of larger systems. The terms "system" and "component" can therefore be used to refer to the same item, but from different viewpoints. It is best to use the former term when speaking of the item and its constituent parts, and the latter term when speaking of the item as a constituent of a larger system.

2.9.2.2 Component, Module

A *component* or *module* is any part of a design that may be instantiated one or more times and combined with other components to form a system or module. A functional component of a system implements a specified function. A hardware component of a system may be a (populated) chassis, board, IC, macro-cell, connector, and so on. A software component or module may be a collection of routines, such as an operating system, or library, as well as a single routine.

Often, the word "component" has been used to specifically designate an IC chip, as in a board component, which indicates a specific structural partitioning. However, since terms such as "IC-chip" adequately describe such devices, it is preferable to not confine the usage of the word component to that single structural level.

Similarly, the word "module" has often been used to designate a multi-chip substrate that is a board component such as the multi-chip module (MCM). However, the word module has also been used to designate larger systems such as the Lunar Module and other non-hardware items such as software code modules. Therefore, the synonyms component and module are used in a recursive and hierarchical terminology that may correspond to either the functional or structural implementation.

Every complex component is itself a system. The terms "system" and "component" (or "module") can therefore be used to refer to the same item, but from different viewpoints. Use the former when speaking of the item and its constituent parts, and the latter when speaking of the item as a constituent of the larger system.

2.9.2.3 Architecture

Following the work being done by the IEEE Architecture Working Group of the Software Engineering Standards Committee [IEEE 1471], an *architecture* is:

"the highest-level conception of a system in its environment."

In particular, the standard comments:

"*An architecture is a property or concept of a system, not merely its structure...any system has an architecture, whether or not it is documented... The phrase 'highest-level' is used to indicate that an architecture abstracts away from details of design, implementation and operation of that system to focus on the system's 'unifying or coherent form'...architecture is not a property of the system alone, but that the system's environment is a consideration in the system's architecture.*"

However, in the context of this book, we use the following three definitions:

- In the context of software, architecture is the configuration of all software routines and services for meeting a system's objective. For example, application, operating system, and communication protocols can describe layers of a software architecture.
- In the context of hardware, architecture is the configuration of all physical elements for meeting a system's objective.
- In the context of systems, architecture is the collection and relationship of the system's constituent hardware and software components. For example, a multiprocessor system's architecture would include the hardware network architecture and the software architecture in the form of distributed and local operating systems, and application and control routines.

An additional concept, related to architecture, that we propose is called a *functional-to-physical architecture mapping*, which is a "mapping" between a functional model of a system and an architecture that implements the functions implied by the functional model. Such a mapping explicitly ties together the main structural components of the system, in both the hardware and software senses, and the functions that are associated with each of them. This mapping should deal with the required communications between functions, and the components in the architecture that are used to realize this communication (for example, in the hardware domain, buses; in the software domain, messaging routines; between hardware and software, memory-mapped I/O). In some contexts, the combination of the architecture of a system–as defined earlier–the functional model of the system, and the

mapping between the two, along with associated implementation constraints, is called an *architecture*.

2.9.2.4 Structure

The following description draws on the IEEE 1471 work and suggestions by JP Calvez:

> *Structure is something arranged in a definite pattern of organization, the organization of parts as dominated by the general character of the whole and the aggregate of elements of an entity in their relationships to each other. Traditionally in electronics and software, "structure is the components and organization of physically identifiable things, whether hardware entities or software objects such as modules, or both."*

2.9.2.5 Hardware

Hardware is a physical component or system implementation of functions. It is especially the intrinsic functional aspects of physical systems that are not electronically modifiable (software-configurable or programmable hardware is simply a software selection of intrinsic hardware functionality). Modification of intrinsic hardware functionality usually does not occur after construction and cannot be changed without physical alteration, since it requires mechanical alteration of circuits. In other words, the intrinsic hardware functionality is set, unless physically altered. Hardware is not intended to change over the unit's life-cycle without physical alteration.

2.9.2.6 Software

Software comprises the electronically modifiable aspects of a system's behavior. This refers especially to aspects that are intended to be changeable multiple times over a unit's life-cycle. Software is often the set of electronically modifiable instruction sequences that are interpreted by hardware and thereby control the operation of the hardware.

2.9.2.7 Firmware

Firmware is a set of electronically modifiable aspects of a system's behavior that is not intended to be altered often, if at all, during a unit's life-cycle. Traditionally, the term has applied to software that is "hard-wired" and cannot be easily changed during the unit life-cycle, but this definition may be too narrow for modern technologies where data can be downloaded and FPGAs reprogrammed while the unit is running.

2.9.3 Information Classes

It is often useful to distinguish between two major types of information that are present in a system: application data and control data. Several other terms depend on distinct definitions for the terms that follow.

2.9.3.1 Application Data

Application data information that is the object of computation or communication, that does not affect, determine, or change the sequence of subsequent operations.

Because decisions as to what to include in a model often pertain to portions of the design defined as control or data, an orthogonal classification must be made to distinguish between control versus data. The challenge in making this classification is that data is control to some and control is data to others; it is "view" specific. For example, all the "signals" used to control a protocol to transfer data over a bus may be viewed as control by a bus designer. But another designer may view any clocked input as data for a synchronous design.

2.9.3.2 Control Data

Control data includes information that affects, determines, or changes subsequent events or operations.

2.9.4 Design Process Terms

2.9.4.1 Synthesis

Design *synthesis* is the process of creating a representation of a system at a lower level of design abstraction from a higher-level (more abstract) representation. The synthesized representation should have the same function as the higher-level representation. Synthesis literally means the combining of constituent entities to form a whole unit. In system design, synthesis refers to the process of finding a set of elements and a way of combining them, such that when so combined to form a system, the system meets its requirements. Synthesis may be automated or done manually, but the term is usually used in reference to an automatic process.

2.9.4.2 Simulation

Simulation is the process of applying stimuli to a model and producing the corresponding responses from the model (when those responses would

occur), in such a way that the responses match the "real-life" expected responses from the system which the model purports to represent.

2.9.4.3 Emulation

Emulation performs the same function as simulation except that a surrogate design is automatically implemented from the design model and programmed into a reprogrammable infrastructure, such as a number of interconnected FPGAs. This emulation model should behave in a functionally identical manner to the model, and to the actual real-life system which the model intends to represent

2.9.4.4 Interface-Based Design

Interface-based design [BAI 00] is a design flow that moves design from an interconnected set of communicating processes with clearly defined and separately captured interface protocols (usually intended to test conceptual behavior) to interconnected realized components in the final system. At this design point, interactions conform to the interface specifications captured at all the levels of abstraction.

At the higher levels of abstraction, the set of operations or tasks required to perform an application are initially linked by "ideal" channels through which information is sent and received as needed, without concern for conflicting resource requests or synchronization. At this stage, the architectural design may be concerned only with functionality or with communication protocols.

As this design is refined, common communication resources are specified, control protocols administered, and sharing of functional units identified. The common issues associated with system design become visible, and the design moves from that of the ideal to the real.

The separate specification of the interfaces allows the design process to proceed fully and concurrently with the minimum of design interference between teams working on separate components.

2.9.4.5 Top-Down Design

Top-down design refers to the flow of design-driving requirements from the abstract function (high-level or top) to the specific implementation (low-level details or bottom). It is the process whereby requirements are developed for the components of a given level of the design-abstraction hierarchy that are used to drive the design and selection of components in the lower levels. In contrast, bottom-up design refers to the process of pre-

selecting certain components, and then partitioning the remaining requirements accordingly.

An example of a top-down design process is one that consists of the following steps:

1. A behavioral level model is partitioned into sub-modules.
2. Interfaces between sub-modules are defined.
3. Resources and requirements for each component module are defined.
4. The process verifies that the partitioned form is equivalent in timing and function to the unpartitioned behavioral model.
5. Steps 1-4 are repeated recursively for each sub-module, until sufficient detail is resolved for physical construction, loading, and operating the system.

During a recursion of steps 1-4, if the verification fails due to an unobtainable requirement, the critical issue is passed upward for reallocation of the requirements. Then the recursion begins again.

Top-down design has in the past been interpreted by some designers literally, as a prescribed and rigid time order of abstraction focus, such that the abstract design would be completed prior to the detailed design in sequence. That interpretation should be avoided because it does not apply to realistic design situations in which top levels could specify unattainable requirements for the lower sub-modules. The preferable interpretation is that multiple levels of the design process are active concurrently, but the flow of requirements is from top to bottom, with feedback on how well the requirements can be met flowing from bottom to top.

2.9.4.6 Prototype

A *prototype* is a preliminary working example or model of a product, component, or system. It is often abstract or lacking in some details from the final version. Two classes of prototypes are used in design processes: physical prototypes and virtual prototypes.

The purpose of a prototype is for testing, exploration, demonstration, validation, and as a design aid. It is used for testing design concepts and exploring design alternatives. Prototypes are also used to demonstrate design solutions or validate design features.

2.9.4.7 Physical Prototype

A *physical prototype* is a physical model of a product, component, or system. The traditional prototype is a physical prototype, as opposed to a virtual prototype. See Section 2.9.4.8, "Virtual Prototype."

Examples of physical prototypes are: bread boards, mock-ups, and brass boards. Physical prototypes are characterized by fabrication times that

typically require weeks to months and take days or weeks to modify. Construction usually involves detailed design, layout, board or integrated-circuit fabrication, ordering, and mounting using solder or wire-wrap. Additionally, programmable systems or parts require detailed target-software design of drivers and operating system, or programming PLAs, FPGAs, and PROMs.

2.9.4.8 Virtual Prototype

A *virtual prototype* is a computer-simulation, or emulation model of a final product, component, or system. Unlike the other modeling terms that distinguish models based on their characteristics, the term virtual prototype does not refer to any particular model characteristic but rather it refers to the role of the model within a design process. A virtual prototype refers to the role of:

- Exploring design alternatives
- Demonstrating design concepts
- Testing for requirements satisfaction and correctness

For more information see Section 2.9.4.6, "Prototype," and Section 2.9.4.7, "Physical Prototype."

Virtual prototypes can be constructed at any level of abstraction and may include a mixture of levels. Several virtual prototypes of a system under design may exist as long as each fulfils the role of a prototype. To be useful in a larger system design, a virtual-prototype model should define the interfaces of the component or system under design.

In contrast to a physical prototype, which requires detailed hardware and software design, a virtual prototype can be configured more quickly and cost-effectively, can be more abstract, and can be invoked earlier in the design process. A distinction is that a virtual prototype, being a computer simulation or emulation, provides greater non-invasive observability of internal states than is normally practical from physical prototypes.

2.9.4.9 Virtual Prototyping

Virtual prototyping is the activity of configuring (constructing) and using (simulating) a computer software-based model of a product, system, or component to explore, test, demonstrate, or validate the design, its concept, or design features, alternatives, or choices. Specifically, this means using the virtual-prototype model as if it were an example of the final (physical) product. For more information see 2.9.4.6, "Prototype" and 2.9.4.8, "Virtual Prototype"

2.9.5 Design-Tool Terms

2.9.5.1 Model

A *model* is the description of a function, system, or component that when executed (usually upon a simulator or emulator), replicates the operation of the intended function on applied stimuli.

2.9.5.2 Emulator

An *emulator* is a hardware device that mimics the electrical behavior at the interfaces of a component, or identified points within a component for in-circuit operation or as a faster means of performing a simulation.

2.9.5.3 Simulator

A *simulator* is a software utility for executing models within a computer. In the case of an HDL simulator, the simulator manages the passage of simulated time and creates the illusion of concurrent model and process execution. The simulator also provides user-interactive or batch-development capabilities such as execution control, which includes break-points, stepping, running, stopping, and continuing; tracing and examining model states; and setting model states.

2.9.6 Verification and Test-Related Terms

For additional terms related to the verification of systems, the reader should also see Chapter 3, "Functional Verification Taxonomy."

2.9.6.1 Testbench

A *testbench* is a model or collection of models or data files that applies stimuli to a device under test (DUT), or a device under verification (DUV), compares the DUT's response with an expected response, and reports any differences observed during simulation.

2.9.6.2 Test Vector

A *test vector* is a set of values for all the external input ports (stimuli) and expected values for the output ports of a module under test.

2.9.6.3 Functional Test

A *functional test* performs tests for the required function of a unit. Functional tests are independent of the implementation of the unit under test. Functional tests do not require implementation knowledge, but test for design errors and correctness. As such, functional tests do not check for physical hardware faults in the manufactured system. For instance, the functional test of a multiplier unit could be 4 * 7 = 28. Such tests check that the unit would perform multiplication and handle corner conditions such as four-quadrant signage.

2.9.6.4 Operational Test

An *operational test* performs tests for the proper operation of a unit. This test is implementation dependent, since it checks for hardware faults such as stuck-at, open, and short. It tests for physical faults in manufactured systems.

2.9.6.5 Boundary Scan

A *boundary scan* is a structured test technique for testing digital circuits. It consists of embedding shift registers at every pin (I/O) of a component so as to control and observe each and every pin independent of the internal logic of the component. Though designers have previously built scan cells in their own ways, IEEE has standardized a test architecture for boundary scan [IEEE 1149].

2.9.6.6 Signature Analysis

Signature analysis is the testing of digital circuits by applying stimuli (a set of inputs) and measuring the response of the circuit (called the *test result*). The result is compared against an expected pattern (called the signature) and fault analysis based on the stimuli (also called a *test vector*). The response is called *signature analysis*.

2.9.7 Requirements and Specifications

2.9.7.1 Specification

A *specification* is any written document or executable program that explicitly states the quantities and functionalities either needed or provided by a system or component. The former class is called *requirements specifications* while the latter are called *design specifications*. Many other types of specifications exist, such as manufacturing specifications, maintenance specifications, and test specifications.

2.9.7.2 Executable Specification (E-Spec)

Traditionally, specifications are a collection of statements written as human-readable documents. The automated form of such statements implemented as executable programming models are known as *E-Specs*. Executable versions of requirements and design specifications, called *ER-Specs* and *ED-Specs* respectively, interact to test a design relative to its requirements.

2.9.7.3 Requirement Specification (Req-Spec)

A *requirement specification* states the necessary and sufficient qualities, quantities, and functions that a system or component must exhibit. Requirements may be expressed in terms of functions, specific values, allowable ranges, or inequalities such as maximums and minimums. Requirement specifications include both electrical behavior of function and timing as seen from the interface, and physical constraints of power, cost, size, fit, and weight.

2.9.7.4 Executable-Requirement Specification (ER-Spec)

Executable versions of Req-Specs are called ER-Specs. They test for requirement compliance by applying tests to candidate systems. In this role, an ER-Spec forms a testbench.

2.9.7.5 Design Specification (Design-Spec)

A *design specification* is the statement of a design solution. The Design-Spec states the requirements for each the system's constituent components and how to configure them as well as the resultant performance, functionality, and other pertinent quantities that characterize the system as designed. The components may be architectural blocks, hardware elements, software elements, or combinations.

2.9.7.6 Executable Design Specification (ED-Spec)

Executable versions of Design-Specs are called ED-Specs. They are used to interact with ER-Specs for automatic requirements testing. An ED-Spec model represents the component or system, while the ER-Spec forms a testbench.

2.9.8 Reusability and Interoperability

2.9.8.1 Reusability

Reusability is the degree to which a module, component, or system may be used again in other instances for which it may or may not have been specifically intended. Reuse occurs across several dimensions, such as life-cycle phases, at the packaging levels, and across model-years. Reuse occurs at various distinct levels, such as:

- Reuse of components (hardware parts) or modules (software object-code), also called *direct implementation*
- Reuse of hardware logic or software source-code recast in new technology or integrated with other logic or code
- Reuse of architecture through re-implementation of functional block concept with new partitioning, integration, or technologies

2.9.8.2 Model Interoperability

Model interoperability designates the degree to which one model may be connected to other models and function properly, with a modicum of effort. Model interoperability requires agreement in interface structure, data format, timing, protocol, and the information content and semantics of exchanged signals.

2.9.9 Interface-Related Terms

2.9.9.1 General Interface Terms

The following set of terms is expected to be common to all component interfaces:

Interface – An *interface* to a component is the sum of all communication–both implicit and explicit–between that component and everything in its environment. It may include not only the static types and sizes of ports, but also the definition of the entire protocol necessary to communicate with a specific instantiation of the component. The interface may define a protocol at many *levels of abstraction*. These levels must be consistent with each other so that the capabilities of the communication protocol observed at one level of abstraction hold at all levels of abstraction below that. An interface to a virtual component consists of a set of *channels* and the protocols defined on these channels. The point at which a channel connects to a component is known as a *port*.

Interface Abstraction Layers/Levels – These are the differing levels of protocol specification that may accompany an interface description. Depending upon the type of interface, certain properties suitable for the description of each layer may be specified. Levels of abstraction on interfaces may be used for:

- Data (for example, from enumeration to bit mask)
- Communication (for example, from point-to-point to bus communication or from transaction level to messages to cells)
- Resource (for example, infinite buffer and non blocking to fixed register and blocking)
- Time (for example, dataflow to serial processes to clocked)

Virtual Component Interface (VCI) – The VCI is the interface of a Virtual Component. It encompasses all the interface abstraction layers from the most abstract layer down to the lowest specified level

Channel – The connectivity mechanism between any two components. Each channel has associated *attributes* along with its behavior. A channel may be specified at multiple levels of abstraction.

Ports – A *port* is a connectable point on the component through which information may travel. A port may have specified attributes and constraints that can range from the direction and size of the port to the definition of the behavioral principle of the port (such as blocking-read) to the specification of the protocol in which the port performs a role.

Protocol – The specification of the communications etiquette. A protocol may include the specification of the control lines and their behavior and relationship to data, the specification of the data types and their values (if necessary), communication timing, state (if implied by the communication semantics), and so on.

Behavioral Blocks – The behavioral entities that correspond to components stripped of their interface protocols are referred to as "behavioral blocks."

Protocol Blocks – The specification of how a set of transport objects and their data combine and cooperate to perform a higher-level task. They can be thought of as "pattern mappings" from one layer of abstraction to another.

Atomic – Defines a property on an interface action. An atomic action either completes fully or not at all. An atomic action that completes fully does so without the possibility of interruption or interference. If an atomic action does not complete fully (that is, it is abandoned) then the state of the system must be as if the action had never started. This is sometimes called *run to completion.*

Datum – A datum is a primitive object that is based on a set of primitive data types (such as "integer," "string," "character," and so on). Data may be

transmitted through interface ports by transport objects such as messages and be acted upon by behavior objects such as protocol blocks.

Cell – A cell is a grouping of zero or more datums (data). Cells may be passed and used like data.

Packet – A packet is a transport object consisting of a group of cells transferred across the component interface.

Operation – An operation is a specialized transport object consisting of a pair of packets, which are usually transferred in different directions, for example, a request packet and a response packet.

Packet Chain – A packet chain is a non-atomic specialized transport object consisting of a set of logically connected packets transferred in the same direction across a component Interface. The chain of packets is connected because no intervening packets are allowed on the same channel.

Message – A message is an atomic transport object that transfers zero or more cells or data in the same direction to or from a port.

Transaction – A transaction is a non-atomic transport object that consists of a set of messages or packet chains across ports and along channels.

Attribute – These are some classifications by which the behaviors of objects can be more specifically defined. For example, a basic transaction may be a *read* and an attribute on that read could be *blocking*.

CHAPTER 3 FUNCTIONAL VERIFICATION TAXONOMY

3.1 Introduction

This section is intended to provide a classification of the various functional verification technologies and uniform definitions of terms used in these technologies.

The intended audience for this section includes design and verification engineers involved in the creation of virtual components (VCs) as well as those engineers who are integrating VCs and verifying SoC designs containing VCs.

3.1.1 Classifications of Verification

A framework for classifying functional verification technologies and methodologies into a logical structure is shown in Table 3-1. It defines four broad categories of verification:

	VC Verification	Integration Verification
Intent Verification	X	X
Equivalence Verification	X	X

Table 3-1 Verification Classification

This section of the taxonomy is organized into these four main categories and includes a detailed description of tools and techniques to implement each type of verification.

- **Intent Verification.** The purpose of this activity is to verify that the designer's intended functionality has been correctly captured in the design. Typically this is done at the highest level of abstraction. The end result establishes a "golden model" that can be used as a

reference for the more detailed design views created throughout the design process.

- **Equivalence Verification**. The purpose of this activity is to verify that the functionality of the various design levels created through the design process matches the functionality of the "golden model."
- **VC Verification.** This is the process of verifying the functionality of a virtual component, that is, unit test.
- **Integration Verification**. This is the process of verifying a system-on-chip (SoC) design that contains one or more VCs, that is, system-level test of the SoC.

Clearly there is a large overlap between the techniques and tools used for each of these four tasks. VC verification and integration verification use the same or similar processes. However, the models and sources of verification test suites may be significantly different. For VC verification, it is critical to verify the detailed functionality of the logic internal to the VC to ensure that the VC was implemented correctly. Integration verification is focused on the interconnection of VCs and their interaction, which can be achieved with models that accurately model the VC interface but approximate the internal functionality.

This document focuses on verifying that a design matches its intended functional behavior as captured in specifications. It does not address high-level issues of validation, such as whether the specification properly captures customer, system-level intent.

3.1.2 Definitions

Term	Definition
Equivalence Verification or Checking	Process of determining whether two designs (which could be of differing levels of abstraction or format) match in terms of functionality; equivalence checking is often performed statically using formal methods
Integration Verification	Process of verifying the functionality of a system-on-chip (SoC) design that contains one or more virtual components
Intent Verification	Process of determining whether a design fulfills a specification of its behavior
Provider Functional	Verification performed by the VC provider

Term	Definition
Verification	
VC Verification	Process of verifying the functionality of a virtual component, for example, unit test of that component

Table 3-2 Verification Definitions

3.2 Intent Verification

The following sections give the tools and techniques that are appropriate for intent verification.

3.2.1 Dynamic Verification

Dynamic verification involves exercise of a model or models of a design, or a hardware implementation of the design, with a set of stimuli. The following dynamic verification tools and techniques are applicable to intent verification.

3.2.1.1 Deterministic Simulation

Simulation is the process of applying stimuli to a model and producing the corresponding responses from that model. In *deterministic simulation*, the stimulus is specified explicitly and an expected response from the model can be predicted and checked. There are two types of simulators: event based and cycle based.

Event-based software simulators operate by taking events, one at a time, and propagating them through a design until a steady state condition is achieved. The design models include the concept of intra-cycle timing as well as functionality. Any change in input stimulus is identified as an event and will be propagated through each stage in the design. A design element may be evaluated several times in a single cycle of each clock due to the different arrival times of the inputs and due to the feedback of signals from downstream design elements. While this provides a highly accurate simulation environment, the speed of execution is dependent upon the size of the design and can be relatively slow for large designs.

Cycle-based simulators take a different approach. Cycle-based simulators have no notion of intra-cycle timing and evaluate the logic between state elements and/or ports in a single shot. Since each logic element is evaluated only once per cycle, this approach significantly reduces the execution time. The simpler model used by a cycle-based simulator (no timing, fewer logic

states, and so forth) can lead to restrictions on the types of circuits that these simulators can handle. For example, circuits that rely on intra-cycle timing or propagation of unknown values for proper simulation may not work with a cycle-based approach.

3.2.1.2 Random Pattern Simulation

In *directed random verification*, random address, data, and control values are driven onto a bus or set of signals, and one or more bus protocol checkers verify that bus protocol violations do not occur as a result of these operations. This verification approach is well suited to bus verification, although random patterns may be useful for simulation of other design structures.

The verification test suites are directed, in that the cycles generated are not purely random but are created to stress the design in specific ways. The pattern generators can be set to create specific transaction types, such as read, write, and read-modify-write in a pseudo-random sequence, but with specific distributions; for example, 20 percent read, 30 percent write, 50 percent read-modify-write. Similarly, data and address fields can be generated in a random sequence, but within specified limits or using a limited set of discrete values. Of course the sequences have to all be valid functions.

These types of verification tests verify corner conditions and sequential or data-dependent situations that are difficult to identify with deterministic simulation. With this methodology, any algorithmic errors are identified and fixed early in the design cycle.

In non-directed simulation, the inputs of a design are driven directly from a random pattern generator and the outputs are checked for any invalid operations. This approach is used most often to verify data-path and arithmetic elements, or to verify small blocks that can accept any random input sequence.

3.2.1.3 Hardware Acceleration

Hardware acceleration is the mapping of some or all of the components in a software simulation into a hardware platform specifically designed for speed up of certain simulation operations. Most commonly, the verification testbench remains running in software while the design being verified is run in the hardware accelerator. Some types of accelerators can also run behavioral code, in which cycle-by-cycle behavior is not fully specified. In this case, it may be possible to run an entire deterministic or random pattern simulation entirely in hardware.

3.2.1.4 Hardware Modeling

Sometimes software simulation models of some design components are unavailable, or are insufficiently accurate. One approach to this dilemma is to run a silicon component in a *hardware modeler*, connected to the software simulator. A hardware modeler receives inputs from the simulator, applies the input to the component and runs one more cycle, then captures outputs of the component and sends them back to the simulator.

3.2.1.5 Monitors

Monitors are probes that watch signals in the design. The probes can be used for various purposes, for example:

- Protocol verification: ensuring that the interface signals obey the protocol defined for the interface
- Performance verification: ensuring that the interface signals obey the performance targets defined for the design (in terms of cycle counts rather than intra-cycle measures or time intervals)
- Recording functional coverage data
- Monitors may be split into two types:
- Interface monitors that only monitor design interface signals
- Internal monitors that only monitor signals internal to the design

Monitors for a VC should be usable at both the VC verification level and at the SoC level (to ensure that the VC is working correctly within the VC integrator's system).

3.2.1.6 Protocol Checkers

Protocol checkers are elements that monitor the transactions on an interface and check for any invalid operations. If an invalid operation is detected in the simulation, it is flagged as an error. These checkers can be embedded in the verification testbench and not be part of the design. For this application, the checkers are active only during simulations. The checkers may also be embedded in the design, where they can actively check for violations not only in the simulation but also during the normal operation of an actual physical device. Checkers embedded in the design should be synthesizable to gates.

3.2.1.7 Expected Results Checkers

An *expected results checker* is part of the system verification testbench that checks the results of a simulation against a previously specified, expected response file. If discrepancies occur, they will be flagged.

3.2.2 Static Functional Verification

Static functional verification exploits formal mathematical techniques to verify a design without the use of verification test suites. There is no industry consensus on the verification approaches included under the static functional verification label. Some consider static functional verification to be a subset of formal verification; others consider them equivalent. Because of this lack of consensus, this document does not consider static functional verification as a distinct technique.

3.2.3 Formal Verification

Formal verification uses mathematical techniques to verify functional aspects of a design. Since formal verification techniques rely on mathematical analysis of the design, verification test suites are not required. The scope of formal verification includes equivalence checking, which is covered in Section 3.3.2. The following formal verification tools and techniques are applicable to intent verification:

3.2.3.1 Property/Model Checking

Property/model checking uses formal mathematical techniques to verify functional properties of designs. A model checker explores the entire state space of a design under all possible input conditions, finding bugs that can be difficult to catch through simulation. When a model checker reports a property to be true, a designer can be 100 percent sure that the report is accurate. Model checking does not require any verification testbench setup. The properties to be verified are specified in the form of queries using a specification language. When the model-checking tool finds an error, the tool generates a complete trace from an initial state to the state where the specified behavior or property failed.

For many designs, only certain input conditions are allowed and therefore only a subset of the entire state space is legal. In this case, a model checker must provide some mechanism by which the designer can specify the allowable input sequences. These are usually specified using a constraint language that describes the bounds for legal input behavior.

Model checking is usually more effective for verifying control-intensive designs than datapath-intensive designs. Systems containing datapaths typically have very large and deep state spaces; verification of properties on such systems can be expensive in memory and processor time. However, property-specific reductions can be used to analyze only that part of the

circuit relevant to the property and design abstraction, which can extend the range of model-checking applications.

Model checkers usually verify properties that a particular condition is always true, can eventually become true, or is never true, under all possible legal input sequences and legal states. Such a property is an assertion about the design, and quite often, is useful in simulation as well as in model checking. Depending upon the mechanism used, it may be possible to use the assertions from simulation directly as properties for model checking.

Usually, assertions state that a particular condition must always be true; if that condition is ever violated in simulation, then the user is notified. When the condition is specified in terms of something that must never happen, this is sometimes called a *checker*. The checker "fires" when the specified condition is violated, and the simulation user is notified.

3.2.3.2 Theorem Proving

Verification systems based on *theorem proving* techniques typically support a specification language based on a chosen kind of formal logic and a set of strategies in the form of commands to mechanically construct a proof of an assertion in the logic. Theorem proving systems widely vary in the kind of formal logic they support and in the level automation they provide for constructing the proof. Most theorem proving systems support a form of universal, general-purpose logic, although there are systems that are customized for specialized or restricted types of logic.

Formal verification of a hardware design, using a verification system based on theorem proving techniques, typically consists of first describing the design model (M) and the property (P) to be verified in the logic/specification language supported by the verification system. The property is verified by constructing a proof of a correctness assertion that M implies P for all possible input conditions. Successful completion of a proof of the correctness criterion guarantees that the property is true of the design for all possible input conditions.

A number of theorem proving systems have been used to perform successful verification of large realistic designs, such as floating point units and complex pipeline control.

As in model checking, verification by theorem proving does not require any verification testbench creation, but requires the formulation of properties to be proved. Unlike model checking, verification by theorem proving is not limited by the size of the inputs or the design state space. Hence, theorem proving is better suited for functional verification of datapath-oriented designs and higher level applications, such as floating point and pipeline control hazard verification. Theorem proving techniques can be used for

property checking, as well as equivalence checking between two models of a design. For checking equivalence between two models, an appropriate assertion relating the two models must be written and proven after describing the two models in the language of the verification system.

The main drawback of verification by theorem proving is that it is not as automatic as model checking, since the user has to construct the proof interactively using the commands of the theorem prover. Another disadvantage is that in the event of a failure to construct a proof, the prover does not automatically construct a counter example trace. The user has to diagnose the cause of the failure by manually analyzing the failed state of the proof.

3.2.4 Dynamic-Formal Hybrid Verification

Some techniques link simulation and formal verification in order to take advantage of the thoroughness of formal techniques while handling larger designs and a wider range of design styles.

3.2.4.1 Symbolic Simulation

Although general purpose theorem proving is too interactive to be widely applicable, certain components of theorem proving systems, such as automatic procedures for deciding restricted types of logic, can be used to build stand-alone verification tools. *Symbolic simulation*, which supports design simulation over symbolic inputs denoting a set of inputs, is one such technology. A symbolic simulation tool uses symbolic values whenever possible, reverting to traditional simulation when required by design size or complexity.

3.2.4.2 Dynamic Formal Verification

Dynamic formal verification uses formal, mathematical methods to amplify or expand design behavior exercised in simulation. Like model checking, it targets assertions by considering a wide range of behaviors that conform to any input constraints. It does not necessarily start from the reset state and consider all possible behaviors for all time. Instead, it starts from a series of states already reached in simulation and explores a range of behavior around that state, usually bounded by sequential depth (the number of clocks). For example, dynamic formal verification may consider all possible five-cycle sequences of legal input changes and associated state transitions from each state in a simulation trace. This technique is optimized for finding ways to violate assertions; it is unlikely to prove that assertions can never be violated.

3.2.4.3 Formal Coverage

Formal coverage, sometimes called *semi-formal verification*, refers to the use of static or dynamic formal methods to improve coverage results as measured by some appropriate metric. Usually, this is accomplished by placing assertions on points not covered in the existing simulation tests, and then targeting these assertions with formal methods. For example, an assertion could be placed inside a basic block in RTL to improve line coverage, or on a state machine to improve arc coverage.

Formal coverage is an emerging verification methodology that is neither widely employed nor directly supported by many formal verification tools.

3.2.4.4 Formal Constraint-Driven Stimulus Generation

Formal constraint-driven stimulus generation is the utilization of formal methods to generate targeted tests that satisfy a given set of constraints. A set of constraints restricts the behavior of a subset of the design input signals over time. The constraints are expressed in a formal specification language (for example, temporal logic). Given the constraints, a stimulus generation tool that uses formal techniques calculates a sequence of stimulus for the design that satisfies the constraints. Depending on the level of integration between the generation tool and the simulator, the stimulus can be directly applied to the design or saved in a format suitable for simulation.

This is an emerging verification methodology that is neither widely employed nor directly supported by many formal verification tools.

3.2.5 Hardware/Software Co-Verification

In the *hardware-software co-verification* methodology, the verification of both the system hardware and software occurs simultaneously. Traditional system design flows occur serially, where the hardware is first fabricated and the system software is then written and debugged on the hardware. With co-verification, the software is executed on the hardware simulation platform while the hardware is being developed, and both hardware and software are debugged in parallel.

Although the creation of the proper co-verification environment can require significant time and expertise, the rewards from using co-verification can be significant. Co-verification allows for many system-level bugs and issues to be uncovered and corrected before the SoC is actually fabricated. Running simulations with the actual processor and firmware code models the system much more accurately and allows for more extensive verification than with simple bus-model transactions used in older design flows. Software is also debugged and verified during the simulations, which allows

system bring-up and development to occur at an accelerated pace when the chip is actually fabricated. Ultimately, co-verification improves the entire product development flow by resolving problems and issues much earlier in the design cycle, saving both time and money.

3.2.6 Emulation

Emulators are specially designed hardware and software systems that are typically built from some type of re-configured logic, often field-programmable gate arrays (FPGAs). These systems are programmed to take on the behavior of the target design and can emulate its functionality, even to the degree of having the emulated design connected directly to the rest of the system in which the design is intended to operate. Since these systems are hardware-based, they can provide circuit simulation speeds that approach the end design target speed. This contrasts with the kilohertz and down to tens of hertz cycle times for software -based simulators. This performance difference of multiple orders of magnitude allows emulation technologies to take on large verification tasks that would take months or even years on a software simulator. Examples of these verification tasks would include applications with very large data sets such as video streams, or with millions of lines of software such as booting up entire operating systems. SoCs with embedded processors often need emulation or prototyping technology to verify the complex functionality of the software running on the embedded processor in conjunction with the surrounding logic prior to committing the design to silicon. As such, these emulation systems are often the common design view between the hardware and software teams in a concurrent design process.

There are many different architectures employed by these emulation systems to provide flexibility, controllability, visibility, and performance. The architectures include arrays of interconnected FPGAs, arrays of custom processors, systems with programmable crossbar switches and programmable bus interface systems. These different architectures provide a range of trade-offs in terms of design capacity, performance, and best-suited design topologies. All of them are intended to work in conjunction with, and to complement, a verification methodology that includes other technologies such as software simulators, timing verifiers, formal verifiers, and logic analyzers.

An emulated design can be viewed in some ways as a prototype with limited accuracy, although it is built from a generic hardware platform. Emulators usually support a high degree of observability into the internal nodes of the design, allowing designs to be debugged in a manner closer to simulation than to actual physical prototypes. In fact, emulators are

sometimes used for simulation purposes, since a software simulator can communicate with an emulator in essentially the same manner as with a hardware simulation accelerator.

Although emulators can sometimes approach the speed of the end design, in some cases their usefulness is limited unless they can run at full speed and connect into the same system as the final design. In addition, the cost of an emulation system usually restricts the number of systems on a project, and this in turn limits the number of engineers who can run emulation at the same time.

3.2.7 Physical Prototyping

A *physical prototype* is a hardware design representation of the target design. This model of the design will operate at "close to" the target platform performance, enabling the following:

- Development and debugging of application and system software before availability of the SoC device
- System-level performance testing
- A high-performance platform simulation for the target design, which enables exhaustive test cycles
- A hardware platform and software environment to support hardware and software co-verification
- A logic analyzer interface for test cases
- Marketing demonstration of the target design

Typically the physical prototype operates within an order of magnitude of the target system speed and is capable of executing at a much higher speed than the software simulators. This means the full software suite can be loaded on the physical prototype and exercised with a system-level verification test suite.

The different approaches for physical prototyping have different characteristics in terms of the amount of design that can be prototyped, the operating speed, and the time to perform changes. The following methods can be used to create a reusable hard prototype for an SoC design:

3.2.7.1 Emulation Systems

This approach adopts an emulation system as defined in Section 3.2.6 as the physical prototyping system.

3.2.7.2 Reconfigurable Prototyping System

This approach maps the VC building blocks of the target design, such as microprocessors, memories, digital signal processors (DSPs), application-specific integrated circuit (ASIC) cores, and I/O interfaces, to off-the-shelf components, bonded-out silicon, non-volatile FPGAs, or in-circuit emulator (ICE) systems. These system components are mounted on daughter boards and plugged into a motherboard that contains custom programmable interconnect devices that can model the connectivity.

3.2.7.3 Application-Specific Prototype

Application-specific prototyping involves developing a complete design that leverages commercially available components and has limited expansion capability. For example, a complete system can be created by interconnecting the Board Support Packages (BSPs) for the target processors and DSPs contained in the design and incorporating the necessary additional components (memories, FPGAs and core devices) to complete the design.

3.2.8 Virtual Prototyping

A *virtual prototype* is a computer simulation model of a product, component, or system. Unlike the other modeling terms that distinguish models based on their characteristics, the term virtual prototype does not refer to any particular model characteristic, but rather refers to the role of the model within a design process. Specifically, a virtual prototype supports the following tasks:

- Exploring design alternatives
- Demonstrating design concepts
- Testing for requirements satisfaction and correctness

Virtual prototypes can be constructed at any level of abstraction and may include a mixture of levels. Several virtual prototypes of a system under design may exist as long as each fulfills one or more of the roles of a prototype. To be useful in a larger system design, a virtual prototype model should define the interfaces of that component or system under design.

In contrast to a physical prototype, which requires detailed hardware and software design, a virtual prototype can be configured quickly and more cost effectively, can be more abstract, and can be invoked earlier in the design process. A distinction is that a virtual prototype, being a computer simulation, provides greater, non-invasion, observability of internal states than is normally practical for a physical prototype. The main drawback of a virtual prototype is that the operating speed is generally much closer to that

of a simulation than to that of a physical prototype, limiting the amount of verification that can be run in reasonable time.

3.2.9 Verification Metrics

The tools and techniques in the following sections may be applied to gauge the completeness of the verification being performed on a design.

3.2.9.1 Hardware Code Coverage

Coverage metrics for verification test suites can be assessed in simulation by using hardware code coverage analysis tools. Code coverage analysis provides the capability to assess some aspects of the functional coverage that a particular verification test suite achieves when applied to a specific design. The analysis tools provide the following:

- A value for the percentage coverage of each attribute being assessed
- A list of unexercised or partially exercised areas of the design

Code coverage analysis is typically performed on the RTL view of the design and assesses the following types of coverage:

- **Statement coverage**: Shows how many times each statement was executed.
- **Toggle coverage**: Shows which bits of the signals in the design have toggled.
- **FSM arc coverage**: Shows how many transitions of the Finite State Machine (FSM) were processed; can be treated as part of path coverage.
- **Visited state coverage**: Shows how many states of the Finite State Machine were entered during simulation.
- **Triggering coverage**: Shows whether each process has been uniquely triggered by each of the signals in its sensitivity list.
- **Branch coverage**: Shows which "case" or "if...else" branches were executed.
- **Expression coverage**: Shows how well a Boolean expression in an "if" condition or assignment has been exercised.
- **Path coverage**: Shows which routes through sequential "if...else" and "case" constructs have been exercised.
- **Signal coverage**: Shows how well state signals or ROM addresses have been exercised.

3.2.9.2 Functional Coverage

Functional coverage is a user-defined metric that reflects the degree to which functional features have been exercised during the verification process. Functional features are either architectural features visible to the user, or major micro-architectural features. Typically, such features cannot be derived automatically from the implementation, and therefore require some specification in the verification testbench.

Functional coverage data is generally the cross-combination between some temporal behavior (for instance, a bus transaction) and some data (such as the transaction source, target and priority). Additional functional coverage information can be obtained by cross-referencing functional coverage points. An example would be the correlation between transactions on two ports of a device, or the correlation between instructions and interrupts in a processor.

Unlike code coverage, functional coverage metrics need to be defined by the developer. A good definition relates closely to the verification plan and covers all major features in the design. Consequently, functional coverage is a much more demanding metric than code coverage. Experience has shown a close correlation between functional coverage and a bugs/week metric.

Functional coverage analysis is typically performed on the RTL view of the design, although some aspects may be assessed at lower-level or higher-level views.

3.2.10 Definitions

Term	Definition
Application-Specific Prototype	A prototype design built from commercially available components
Assertion Monitors	Monitors to check that properties of the design hold during dynamic simulation; also called assertion checkers
Assertion or Property	A statement of design intent that can be checked in dynamic simulation and formal verification
Behavioral Model	Model that exhibits some or all of the functionality of the artifact being modeled but is not written to be taken through a design flow and therefore may not be synthesizable)
Branch Coverage	Measures which branches were executed, for example, "case" or "if...else" branches

Term	Definition
Code Coverage	Coverage metrics defined in terms of syntax of the design and measured during dynamic simulation of the design, including statement coverage, toggling of variables, finite state machine transition and state visitation coverage, if-else branch coverage, conditional statement coverage (metrics on the ways that a condition can become true), paths through if-else and case statements and general signal coverage
Compliance Tests	Tests provided to demonstrate that a design complies to some agreed standard (such as the AMBA bus specification protocol)
Constraint	Rules defining relationships between signals within a design; they can be combinatorial or sequential/temporal and can be used in pseudo-random generation and formal methods, for example
Coverage Monitors	Monitors that checks that certain events occur in the design during dynamic simulation
Cycle-Based	A simulation in which each element of a design is evaluated only once per clock cycle; can be contrasted to event-based simulation
Directed or Deterministic Simulation	Simulation in which the stimulus is specified explicitly, as is the expected response of the design model; can be contrasted with random or pseudo-random simulation
Driver	The part of the testbench that drives values onto the signals of the design being verified; it is considered good testbench design style that they only drive interface signal
Dynamic Formal Verification	Techniques which use simulation results as the starting point for formal techniques, typically exploring only a portion of a state space and are therefore less exhaustive than model checking
Dynamic Verification	Execution of a model or models of a design with a set of stimulus

Term	Definition
Emulation	Specially designed hardware and software systems, using some re-configurable hardware, such as FPGAs, that can simulate a hardware design faster than conventional workstation or PC-based simulators
Event-Based Simulation	A simulation in which events (changes of input valuations, which may occur multiple times during a clock cycle) are propagated through a design until a steady state condition results; can be contrasted to cycle-based simulation.
Expected Results Checkers	A means for checking the results of a simulation against a previously specified, correct response
Expression Coverage	Measures how well a boolean expression has been exercised, for example, the boolean expression used in an "if" condition
Formal Constraint-Driven Stimulus Generation	The use of formal methods to generate targeted tests that satisfy a set of constraints
Formal Coverage or Semi-Formal Verification	The use of static or dynamic formal methods to improve coverage results as measured by some appropriate metric
Formal Verification	The use of mathematical techniques and formalisms to verify aspects of a design, spanning both intent and equivalence verification; such techniques are often called static because they do not involve execution of the design and can be contrasted to dynamic verification
FSM Arc Coverage	Shows how many transitions of a Finite State Machine (FSM) were executed
Functional Coverage	Coverage metrics, generally related to behavior that changes over time and sequences of events such as tracking of bus interactions; these need to be individually defined by people knowledgeable about the design and its intent
Hardware Acceleration	A system for mapping all components of a software simulation onto a hardware platform that is specifically designed to speed up the simulation process

Term	Definition
Hardware Modeling	A system in which a simulator receives input from and sends output to a hardware component
Hardware/Software Co-Verification	A system in which the hardware and the software portions of a design are executed and verified in parallel
Input Constraint	A constraint on input signals
Model Checking or Property Checking	A formal verification technique for checking the entire state space of a design for violations of properties, for example, specifications of behavior
Monitor	Monitors are probes that observes signals in the design during dynamic simulation
Path Coverage	Shows which routes through sequential "if...else" and "case" constructs have been exercised
Physical Prototyping	A hardware representation of a design (often created using FPGAs) that operates at speeds close to, but not necessarily as fast as, the ultimate design to be built
Protocol Checkers	A means for checking behavior of an interface and determining if violations of defined, acceptable behavior have occurred
Pseudo-Random Simulation	A dynamic simulation technique where the design is stimulated with pseudo-random inputs by the user exercising some control over the random stimulus generation; can be contrasted with directed and random simulation
Random or Non-Directed Simulation	A dynamic simulation technique where the design is stimulated with random inputs; can be contrasted to directed and pseudo-random simulation.
Reconfigurable PrototypingSystem	A system in which the VCs of an SoC design are created in off-the-shelf components, bonded-out silicon, FPGAs or in-circuit emulator systems
Register Transfer Language (RTL)	A programming language representation of a design in which some, but not all of the design structure is explicitly represented, such as the Verilog and VHDL languages

Term	Definition
Signal Coverage	Shows how well state signals have been exercised
Statement Coverage	Shows how many times a statement in the RTL was executed
Static Functional Verification	Exploitation of formal mathematical techniques to verify a design without the use of verification test suites; there is no industry consensus on the verification approaches included under this term
Symbolic Simulation	Simulation in which some or all inputs are symbolic variables, and functions of these variables are propagated through a design
System-on-Chip (SoC)	A single piece of silicon containing multiple VCs to perform a certain defined function
Testbench	The overall system for applying stimulus to a design and monitoring the design for correct responses and functional coverage
Theorem Proving	A formal verification technique in which a specification is expressed in a formal logic and proof strategies are utilized to construct a proof that a design obeys the specification
Toggle Coverage	Shows which bits of the signals in the design have toggled
Triggering Coverage	Shows whether each process has been uniquely triggered by each of the signals in its sensitivity list
Verification Metrics	Techniques for measuring the effectiveness of verification procedures on a design; these include code coverage metrics, functional coverage metrics and bug-tracking metrics
Virtual Prototyping	A simulation model of a component or an entire system, useful for exploring design alternatives and testing for correctness
Visited State Coverage	Shows how many states of a Finite State Machine (FS M) were entered during simulation

Table 3-3 Intent Verification Definitions

3.3 Equivalence Verification

As a design progresses through the development process, abstract models of the design are refined with greater levels of functional detail. Each of these functional views should be verified against the original design intent. This verification is called *equivalence verification*. For SoCs with both hardware and embedded software content, both the hardware and software are refined and require equivalence verification. Software refinement consists of code optimizations for either performance or code size. These optimizations may be realized by modification of parameters in a code generation tool, through manual optimizations at the language level, or by the replacement of critical code portions with assembly level optimizations. Each of these refinements must be revalidated on an appropriate model of the hardware. The functional verification mapping table in Section 3.6 shows different options for hardware models that may be used for software equivalence verification.

The following tools and techniques are appropriate for hardware equivalence checking.

3.3.1 Dynamic Verification

The following dynamic verification tools and techniques are applicable to equivalence verification:

3.3.1.1 Deterministic Simulation

As described in Section 3.2.1.1, *deterministic simulation* is the process of applying explicitly specified stimuli to a model, producing the corresponding responses from that model, and comparing the simulated and expected responses. Once a verification testbench and verification test suite have been developed for an RTL model, the same set of verification tests can be simulated using a gate-level netlist for the same design to check whether the results are the same. In some cases, it may also be possible to run the same verification tests in simulation on higher-level (such as, behavioral) or lower-level (such as, switch) models of the same design.

3.3.1.2 Expected Results Checkers

Expected results checkers check the results of a simulation against a previously specified, expected response file. The checker flags any mismatches that occur.

3.3.1.3 Golden Model Checkers

Golden model checkers monitor the responses of two models of the design, compare the responses to the input stimuli and flag any discrepancies. One model is the "golden" or trusted model and the other is the unproven design being verified. Generally, the comparison does not involve any formal verification techniques; the responses of the two models are simply compared upon any change.

3.3.1.4 Regression Testing

Running *regression tests* on a design generally implies two attributes of the verification environment. The first is an automated verification setup in which all required electronic design automation (EDA) tools, verification testbenches and results analysis can be run in batch mode. This process makes it possible for the regression tests to run in the background on compute servers with minimal human intervention.

The second feature of a true regression test is that its success or failure can also be determined in batch mode. This makes it possible for an engineer to determine the regression results simply by examining a log file. Success or failure is usually determined by comparing the regression test results with a set of golden results from a prior regression run. Therefore, regressions can be viewed as a form of equivalence checking.

Regression tests are most often used with deterministic simulation and also with random pattern simulation, when the random behavior is identical from run to run. However, most types of functional verification can also be run in regression, including formal and semi-formal verification. Regressions are most commonly used to verify that a design change does not cause any existing verification tests to fail. Regression test suites tend to grow incrementally as new verification tests and the design itself evolves.

3.3.1.5 Verification Test Suite Migration

Applying a system level verification test suite to other views of the design requires the ability to migrate or transpose the suite into a format suitable for application to RTL and netlist views of the design. The basic approach to migrating a verification test suite from one level of the design to the next is as follows:

- Translate the stimulus from the upper level to a format suitable for application at the next lower level.
- The verification test suites can be applied to both versions of the design and the results compared at points of divergence between the designs.

- A new version of the verification test suite can be extracted from the lower level model that contains the additional level of detail provided by the model.

The following paragraphs describe how migration between levels may be achieved. To ease the migration of verification test suites from the functional level to lower levels, certain restrictions are applied to the functional verification test suite:

- The verification test suite should use bit true representations for data. While data values in "C" have no concept of bus width, hardware implementations of these functions will have a fixed bus width. Modeling at the functional level using bit true representations will ensure convergence of results.
- The same arguments apply to the use of fixed-and floating-point representations. Fixed-point implementation should be used for functional modeling. This will aid in the alignment of the functional model and hardware implementation.

Functional to RTL Migration

The system level functional design is usually written in "C" or behavioral hardware description language (HDL) and the associated verification test suite is token-based or block-based. A token in this instance is a data block of arbitrary size. The functional model has no concept of time or clocks and the verification test suite is applied by an event scheduler. Typically the results of this verification test suite will be written to an external memory and the success or failure of the verification test will be determined by the memory contents on completion of the verification test.

To migrate this verification test to an RTL level model, the tokens must be translated into pin-and bus-level cycles with the associated clocks. The results are checked by comparing the external memory contents created by running the functional test on the functional model with the "migrated" test run on the RTL. Once they match, at the points of disagreement between these models, a new verification test suite can be created by capturing the cycle by cycle behavior of the RTL model. This cycle by cycle behavior may be captured at the I/Os of the design or may include internal state traces. This new verification test suite may then be used for comparing the RTL to lower design abstractions.

RTL to Netlist Migration

The register transfer level (RTL) to netlist migration is achieved by transforming the verification test suite created from the RTL model into a

format suitable for application to the netlist level. The bus-based RTL verification test suite is translated into a bit and pin accurate stimulus. This stimulus can then be applied to the netlist model and the results compared with the RTL response at the points of disagreement. In this case the points of divergence are the I/O pins and internal state elements. These comparison points are sampled at the end of each cycle. Once these points have been verified as matching, a more detailed verification test suite can be created by capturing the switching times within a cycle for the output transitions.

3.3.2 Formal Equivalence Checking

Formal equivalency checking tools verify that two views of a design are functionally equivalent as viewed at the I/O boundaries and on a cycle-by-cycle basis. Formal equivalency checking is usually applied to RTL and gate level netlists of a design, and in some cases, can be applied to high-level or lower-level models as well.

There are several advantages offered by formal equivalency checking over simulation. Formal equivalency checking provides complete equivalency checking as opposed to simulation, which verifies equivalency only to the extent that the verification test suite exercises the design. It can execute in a fraction of the time that an exhaustive simulation would run. It helps automate the verification and debug of errors. Equivalence checkers usually provide detailed "counter-examples" that demonstrate the mismatches down to individual logic cones.

3.3.2.1 Boolean Equivalence Checking

Most tools for equivalence checking are *Boolean equivalence checkers*, which means they check combinational logic. With such tools, name mappings are made between memory elements (flip-flops, latches, and so forth) in each of the two design formats being compared, usually automatically. When the mapping has been determined, the tool then checks that the combinational logic function at the input to each pair of name mapped memory elements is equivalent. This means that for each possible combination of inputs, the combinational logic outputs, which are the inputs to the memory elements, are the same.

3.3.2.2 Sequential Equivalence Checking

It could be the case that two designs have different numbers of, or different arrangements of, memory elements, but are still equivalent in the sense of producing the same input-output streams, given some alignment of initial states between the two machines. This is termed *sequential*

equivalence. An example would be two implementations of a finite state machine, where one implementation is fully encoded, such as, using 3 latches to encode 8 states, and the other is one-hot, such as, using 8 latches to encode 8 states, and yet both machines have the same output, starting from initial states, for the same input streams.

Sequential equivalence checking presents a much harder problem to solve than Boolean equivalence checking. Therefore, there have been few tools available for this task. Many Boolean equivalence checkers do have some support for sequential equivalence checking, allowing small finite state machines to be checked (usually, the user must supply, by hand, a mapping of assumed equivalent state assignments, which limits this to only small machines) or simple movements of certain combinational logic devices across memory element boundaries to be checked. But, the general, sequential equivalence checking of large designs remains an unsolved problem, from a practical point of view.

3.3.3 Physical Verification

Physical verification is the process of checking the geometric design database to ensure that the physical implementation is a correct representation of the original logic design. Physical verification consists of three distinct checks: Electrical Rules Checks (ERC), Design Rules Checks (DRC) and Layout Versus Schematic Checks (LVS). The standard geometric database format is GDSII-Stream. The GDSII-Stream database for a design contains a polygon representation of the circuit, separated into the different design layers for the target process.

ERC refers to the procedure of checking the geometric database for electrical design rule violations. These electrical design rules are process specific and include checks for unused outputs, floating inputs, and loading violations. Connectivity violations, such as open and shorts, are also checked.

DRC refers to the procedure of checking the geometric database against the process design rules. These rules are gathered in a DRC rules file and include checks such as layer-to-layer spacing, trace widths on a specific layer, layer-to-layer overlaps, and so on.

LVS refers to the procedure of checking the geometric database against a "golden" netlist. The LVS tool constructs a netlist by extracting polygons and building devices from the physical layout. This extracted netlist is then compared to the "golden" netlist. All devices and interconnect must match exactly. Physical verification is performed on the geometric database prior to release for mask generation and fabrication.

Additional forms of physical verification that affect timing, such as signal integrity, crosstalk, metal migration, and noise, fall outside the scope of the present document and its focus on functional verification.

3.3.4 Definitions

Term	Definition
Behavioral Model	Model that exhibits some or all of the functionality of the artifact being modeled but is not written to be taken through a design flow and therefore may not be synthesizable)
Boolean or Structural Equivalence Checking	A formal equivalency check in which pairings of inputs, outputs and memory elements are first determined for each of two design versions, and then the combinational cone of logic at the inputs to each memory element or output of a pair is proven equivalent, meaning it is proven that the same truth table is implemented
Formal Equivalence Checking	The application of formal methods to equivalence verification; Boolean/structural and sequential equivalence are two examples of formal equivalence checking
Functional to RTL Test Suite Migration	A means for translating a test suite for an abstract behavioral model of a design into a test suite suitable for the RTL level
Golden Model Checkers	Simulation monitors that check the responses of two models of a design, one of which is considered the reference or "golden" model
Physical Verification	The process of checking the geometric design database to ensure that the physical implementation is a correct representation of the original logic design, consisting of three distinct checks: Electrical Rules Checks (ERC), Design Rules Checks (DRC) and Layout Versus Schematic Checks (LVS)

Term	Definition
Regression Testing	Techniques for running large numbers of "verifications" (such as tests and property checks) in batch mode, with minimal human intervention, with results analyzed in batch mode and pass/fail outcomes reported in an automatic way
RTL to Netlist Test Suite Migration	A means for translating a test suite that operated on the RTL level to one that operates on the netlist level of a design
Sequential Equivalence Checking	Formal equivalency checking techniques that require mapping of inputs and outputs but do not rely on mapping of memory elements in one design to another, but rather prove, given a set of initial states for each of the designs, that designs with different numbers of, or different arrangements of, memory elements produce the same output streams given the same input streams
Stub Model	A particular type of behavioral model that only models the interface signals to allow connectivity to be tested. Outputs may be assigned values in the stub model
Verification Test Suite Migration	A means for translating a test suite that operated on one design level (for example, gate netlist) to another level such as RTL

Table 3-4 Equivalence Verification Definitions

3.4 VC Verification

VC verification encompasses both intent verification and equivalence verification. Intent verification may be done at any level. However, it should be noted that the higher the level of abstraction applied, the more efficient will be the resulting verification. Today, intent verification is typically done at the RTL level, since RTL is the highest level of abstraction handled by many tools (model checking, equivalence checkers, code coverage, and so forth) used in the design process.

Equivalence verification, as a minimum, must validate that the lowest level of design decomposition (GDSII-Stream, netlist, and so forth) is

verified against the golden model (previously verified during the intent verification phase). Typically many, if not all, intermediate design abstractions will be verified.

3.5 Integration Verification

Integration verification encompasses both intent verification and equivalence checking and is aimed at verifying an SoC that incorporates one or more VCs. Since the intent is to verify the integrated SoC and not the individual components, "gray box" models can be used for the verification. These accurately model the VC's interfaces, but not the internal function. While integration verification does not seek to duplicate the VC verification executed by the VC provider, the integrator may want to review the VC verification plan to qualify/certify the VC.

Since the integration verification may stimulate the VC in ways not anticipated by the VC provider, verification with a "white-box" model that has full internal functionality has the potential to detect errors in the VC itself.

3.6 Functional Verification Mapping

Table 3-5 maps the various functional verification models and technologies against the different verification steps. For intent verification, deterministic simulation, random pattern simulation, protocol checking, and expected results checking are grouped together under simulation. Similarly, for equivalence checking, deterministic simulation, expected results checkers, golden model checkers, and verification test suite migration are grouped together under simulation.

In this table, the models associated with a specific verification technology may be used either directly or indirectly. An example of the direct use of a model is simulation where the functional/behavioral/RTL models are directly used by the simulator. An example of indirect use is in physical prototyping where a functional model is used to define the required functionality of the prototype and is then mapped into physical devices with equivalent functionality.

Verification Step	Verification Technology	Models
Hardware Intent	Simulation	Functional
		Behavioral
		RTL
	Emulation	RTL
	Model Checking	RTL
	Theorem Proving	RTL
	Physical Prototype	Behavioral
		RTL
		Logic
	Virtual Prototype	Functional
	Code Coverage	RTL
Software Intent	Hardware/Software Co-Verification	Behavioral
		RTL
	Emulation	RTL
	Physical Prototype	Behavioral
		RTL
		Logic
Hardware Equivalence	Simulation	Behavioral
		RTL
		Logic
		Gate
		Switch
		Circuit
	Emulation	RTL
	Equivalence Checking	RTL
		Gate

Table 3-5 Functional Verification Mapping

3.7 Summary

Functional verification is a complex topic that is further complicated by the fact that many tools and models are used for multiple tasks within a verification process. Table 3-6 summarizes the definitions provided in this taxonomy to provide a common language for verification models, tools, and techniques.

Term	Definition
Application-Specific Prototype	A prototype design built from commercially available components
Assertion monitors	Monitors to check that properties of the design hold during dynamic simulation; also called assertion checkers
Assertion or property	A statement of design intent that can be checked in dynamic simulation and formal verification
Behavioral Model	Model that exhibits some or all of the functionality of the artifact being modeled but is not written to be taken through a design flow and therefore may not be synthesizable)
Boolean or Structural Equivalence Checking	A formal equivalency check in which pairings of inputs, outputs and memory elements are first determined for each of two design versions, and then the combinational cone of logic at the inputs to each memory element or output of a pair is proven equivalent, meaning it is proven that the same truth table is implemented
Branch Coverage	Measures which branches were executed, for example, "case" or "if...else" branches
Bus functional model or BFM	Used to provide simplified bus agent models for verifying designs that attach to buses (such as the AMBA bus)

Term	Definition
Code Coverage	Coverage metrics defined in terms of syntax of the design and measured during dynamic simulation of the design, including statement coverage, toggling of variables, finite state machine transition and state visitation coverage, if-else branch coverage, conditional statement coverage (metrics on the ways that a condition can become true), paths through if-else and case statements and general signal coverage
Compliance tests	Tests provided to demonstrate that a design complies to some agreed standard (such as the AMBA bus specification protocol)
Constraint	Rules defining relationships between signals within a design; they can be combinatorial or sequential/temporal and can be used in pseudo-random generation and formal methods, for example
Coverage monitors	Monitors that checks that certain events occur in the design during dynamic simulation
Cycle-Based Simulation	A simulation in which each element of a design is evaluated only once per clock cycle; can be contrasted to event-based simulation
Directed or Deterministic Simulation	Simulation in which the stimulus is specified explicitly, as is the expected response of the design model; can be contrasted with random or pseudo-random simulation
Driver	The part of the testbench that drives values onto the signals of the design being verified; it is considered good testbench design style that they only drive interface signals
Dynamic Formal Verification	Techniques which use simulation results as the starting point for formal techniques, typically exploring only a portion of a state space and are therefore less exhaustive than model checking
Dynamic Verification	Execution of a model or models of a design with a set of stimulus

Term	Definition
Emulation	Specially designed hardware and software systems, using some re-configurable hardware, such as FPGAs, that can simulate a hardware design faster than conventional workstation or PC-based simulators
Equivalence Verification or Checking	Process of determining whether two designs (which could be of differing levels of abstraction or format) match in terms of functionality; equivalence checking is often performed statically using formal methods
Event-Based Simulation	A simulation in which events (changes of input valuations, which may occur multiple times during a clock cycle) are propagated through a design until a steady state condition results; can be contrasted to cycle-based simulation.
Expected Results Checkers	A means for checking the results of a simulation against a previously specified, correct response
Expression Coverage	Measures how well a boolean expression has been exercised, for example, the boolean expression used in an "if" condition
Formal Constraint-Driven Stimulus Generation	The use of formal methods to generate targeted tests that satisfy a set of constraints
Formal Coverage or Semi-Formal Verification	The use of static or dynamic formal methods to improve coverage results as measured by some appropriate metric
Formal Equivalence Checking	The application of formal methods to equivalence verification; Boolean/structural and sequential equivalence are two examples of formal equivalence checking
Formal Verification	The use of mathematical techniques and formalisms to verify aspects of a design, spanning both intent and equivalence verification; such techniques are often called static because they do not involve execution of the design and can be contrasted to dynamic verification
FSM Arc Coverage	Shows how many transitions of a Finite State Machine (FSM) were executed

Term	Definition
Functional Coverage	Coverage metrics, generally related to behavior that changes over time and sequences of events such as tracking of bus interactions; these need to be individually defined by people knowledgeable about the design and its intent
Functional to RTL Test Suite Migration	A means for translating a test suite for an abstract behavioral model of a design into a test suite suitable for the RTL level
Golden Model Checkers	Simulation monitors that check the responses of two models of a design, one of which is considered the reference or "golden" model
Hardware Acceleration	A system for mapping all components of a software simulation onto a hardware platform that is specifically designed to speed up the simulation process
Hardware Modeling	A system in which a simulator receives input from and sends output to a hardware component
Hardware/Software Co-Verification	A system in which the hardware and the software portions of a design are executed and verified in parallel
Input constraint	A constraint on input signals
Integration Verification	Process of verifying the functionality of a system-on-chip (SoC) design that contains one or more virtual components
Intent Verification	Process of determining whether a design fulfills a specification of its behavior
Model Checking or Property Checking	A formal verification technique for checking the entire state space of a design for violations of properties, for example, specifications of behavior
Monitor	Monitors are probes that observes signals in the design during dynamic simulation
Path Coverage	Shows which routes through sequential "if...else" and "case" constructs have been exercised

Term	Definition
Physical Prototyping	A hardware representation of a design (often created using FPGAs) that operates at speeds close to, but not necessarily as fast as, the ultimate design to be built
Physical Verification	The process of checking the geometric design database to ensure that the physical implementation is a correct representation of the original logic design, consisting of three distinct checks: Electrical Rules Checks (ERC), Design Rules Checks (DRC) and Layout Versus Schematic Checks (LVS)
Protocol Checkers	A means for checking behavior of an interface and determining if violations of defined, acceptable behavior have occurred
Provider Functional Verification	Verification performed by the VC provider
Pseudo-random Simulation	A dynamic simulation technique where the design is stimulated with pseudo-random inputs by the user exercising some control over the random stimulus generation; can be contrasted with directed and random simulation
Random or Non-Directed Simulation	A dynamic simulation technique where the design is stimulated with random inputs; can be contrasted to directed and pseudo-random simulation.
Reconfigurable Prototyping System	A system in which the VCs of an SoC design are created in off-the-shelf components, bonded-out silicon, FPGAs or in-circuit emulator systems
Register Transfer Language (RTL)	A programming language representation of a design in which some, but not all of the design structure is explicitly represented, such as the Verilog and VHDL languages
Regression Testing	Techniques for running large numbers of "verifications" (such as tests and property checks) in batch mode, with minimal human intervention, with results analyzed in batch mode and pass/fail outcomes reported in an automatic way

Term	Definition
RTL to Netlist Test Suite Migration	A means for translating a test suite that operated on the RTL level to one that operates on the netlist level of a design
Sequential Equivalence Checking	Formal equivalency checking techniques that require mapping of inputs and outputs but do not rely on mapping of memory elements in one design to another, but rather prove, given a set of initial states for each of the designs, that designs with different numbers of, or different arrangements of, memory elements produce the same output streams given the same input streams
Signal Coverage	Shows how well state signals have been exercised
Statement Coverage	Shows how many times a statement in the RTL was executed
Static Functional Verification	Exploitation of formal mathematical techniques to verify a design without the use of verification test suites; there is no industry consensus on the verification approaches included under this term
Stub model	A particular type of behavioral model that only models the interface signals to allow connectivity to be tested. Outputs may be assigned values in the stub model
Symbolic Simulation	Simulation in which some or all inputs are symbolic variables, and functions of these variables are propagated through a design
System-on-Chip (SoC)	A single piece of silicon containing multiple VCs to perform a certain defined function
Testbench	The overall system for applying stimulus to a design and monitoring the design for correct responses and functional coverage
Theorem Proving	A formal verification technique in which a specification is expressed in a formal logic and proof strategies are utilized to construct a proof that a design obeys the specification
Toggle Coverage	Shows which bits of the signals in the design have toggled

Term	Definition
Triggering Coverage	Shows whether each process has been uniquely triggered by each of the signals in its sensitivity list
VC Verification	Process of verifying the functionality of a virtual component, for example, unit test of that component
Verification Metrics	Techniques for measuring the effectiveness of verification procedures on a design; these include code coverage metrics, functional coverage metrics and bug-tracking metrics
Verification Test Suite Migration	A means for translating a test suite that operated on one design level (for example, gate netlist) to another level such as RTL
Virtual Prototyping	A simulation model of a component or an entire system, useful for exploring design alternatives and testing for correctness
Visited State Coverage	Shows how many states of a Finite State Machine (FSM) were entered during simulation

Table 3-6 Summary of Definitions

CHAPTER 4 PLATFORM-BASED DESIGN

4.1 Platform-Based Design

Migration from boards and boxes to Systems-on-Chips (SoCs), consisting of embedded software and a variety of computing and other hardware components, is now a common practice. Indeed, many new designs are conceived of as SoC based products from scratch. Due to the costly design, integration, processing, and testing phases in the SoC life cycle, the industry is interested in any development approach that may lead to greater efficiency and lower costs. One such approach is platform-based design (PBD), where integrated and verified platforms serve as the basis for families of derivative products.

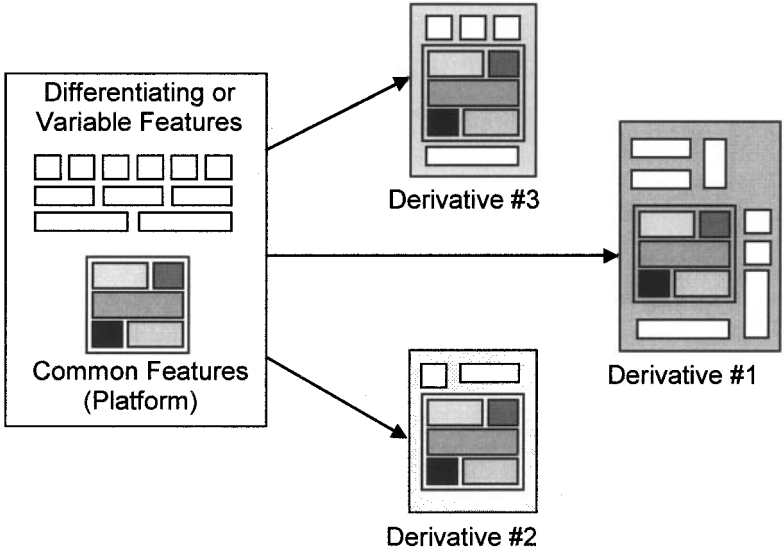


Figure 4.1 The PBD Process

Figure 4.1 shows the basic PBD process: common features supporting multiple higher-level products are aggregated into a platform, then integrated

with product, application, or domain-specific features (often described as "differentiating IP" or "variable features") to form one or more derivative products. Depending on the developer's needs, a product family of multiple derivatives can either be developed simultaneously, or derivatives can succeed each other over time. A platform may be a "black box," in that it is characterized by its external interfaces; its internal contents cannot be changed by its user, though some attributes may be configurable. Unlike some design approaches, in PBD each product is a derivative of the platform, not the reconstruction of an earlier product. Also, PBD can be used at any level of abstraction or construction in building a system.

Modeling of the complete system or of specific aspects and components at various levels of abstraction has been addressed in earlier chapters of this book. High-level models allow us to specify and verify system requirements; to analyze, explore, compare, and select different components of a system; and to explore several architectural choices. An essential element of efficient design practice is the ability to apply principles of system-level design to identify, scope, and design platforms based on reusable entities called *Virtual Components* (VCs). VCs are design or verification objects, including hardware, software, verification, and model components, and subsystems consisting of some or all of these, specifically encapsulated for reuse in a coordinated, managed form. VCs can also be hierarchical and inclusive: a subsystem is itself a VC, once it has been encapsulated for reuse.

It should be noted that the platform concept is applicable at higher levels of system than just SoC. Indeed, platforms have been used at the board and subsystem levels in many branches of electronics design for many years. From one perspective, a computing "platform" such as the IBM 360 or the PC platform (as discussed in [SAN 01 and 02]) has acted like this both for various hardware-software derivatives and as a delivery vehicle for software-based applications. Thus, for a virtual component, one can think of "component" or design blocks. However, in this chapter we will use the term VC and virtual component–but the reader may keep in mind the wider use of these concepts.

The goal of this chapter is to assist the design community of system designers and integrators, software developers, and hardware design teams–both the SoC builder and the design team integrating an SoC into a higher level product–to evaluate and define platform features and architectures, and sets of variable or differentiating VCs, to use in construction of a family of related, platform-based products. The taxonomy and concepts contained herein have been based on a consensus that developed on the meanings of common PBD-related terms and classifications acceptable to the appropriate design community. The PBD approach is fully scalable, meaning it applies in essentially the same way to development of both SoC-based systems and

the SoC itself. During the process of developing these concepts, if conflicting meanings existed in the different communities involved, the most common definitions were chosen, or an enveloping definition was created

4.1.1 Introduction

As technology advances, the business pressure to design large SoCs in a short time increases. Design reuse is a prevalent method for improving design efficiency of large SoCs. In many cases, reused blocks are internally developed. However, even with the rapid advances in fabrication technology and design tools, few companies can dedicate the resources necessary to design and maintain all of the blocks required to offer the customer a total system solution.

Consequently, it has become critical for companies to both increase their access to a variety of intellectual property (IP) blocks, both hardware and software, and to make more efficient use of them, to meet their time-to-market and business objectives. By doing so, each company can focus their limited design resources on areas where they provide maximum value, while using industry-wide design expertise to produce IP to satisfy their needs.

The goal of accelerating design reuse of IP has been achieved by addressing the challenges of design in a divide-and-conquer approach, breaking down the entire design reuse problem into component pieces and attacking and resolving each individually in a pragmatic, market-driven approach. Platform-based design extends this approach by identifying common hardware and software features that can be reused in many products within a product line or product family, and aggregating them into a platform. The platform is thus an integrated subsystem or VC that can be reused as the basis for multiple derivative products, the product family members.

4.1.2 Background and History

A crucial requirement for this taxonomy is that it be useful for characterizing, selecting, using, and building integratable platforms in an SoC design process. The terminology is based on the commonly documented and applied vocabulary in the digital electronic design and modeling industry, and it draws heavily from related previous and ongoing efforts by many groups and individuals.

Effective use of product platforms and product families in technology industries has been discussed in many places. Some of the best general discussions are those by Meyer and Lehnerd [MEY 97]; Gawer and Cusumano [GAW 02], and the classic work by Davidow [DAV 86]. In the

specific area of SoC platforms, the recent books by Chang, Cooke, Hunt, Martin, McNelly, and Todd [CHA 99] and by Martin and Chang [MAR 03], and papers and presentations by Sangiovanni-Vincentelli [SAN 01, 02], Ferrari [SAN 99], and Keutzer et al. [KEU 00], are key. We also look to the work done in the Software Product Lines research community, particularly the US-based Software Engineering Institute (SEI) Software Product Line Practice initiative [SEIPLP], and the European Software Institute (ESI) ESAPS [ESAPS] and CAFÉ [CAFE] programs, as well as work done at TrueScope Technologies [TRUE], and at the Fraunhofer Institute.

With the participation of researchers from several companies, these programs have made numerous contributions to the body of knowledge of product families and product lines, system architecting, development of common platforms and variable differentiating features, and the construction of multiple variant or derivative products within a product family.

4.1.3 Platform-Based Development System

A complete approach to PBD involves more than just specification of platform hardware and software features and VCs. To be usable, a platform-based development system (PBDS) must include complete support for business, tool, and support practices in addition to its defined features and components. Table 4-1 shows the properties of a complete PBDS (the list shown here is extensive, but not complete or exhaustive). The PBDS described here is provided by platform developers for their customers, the platform integrators, to use.

Business and Economics	**Components and Features**
Product line business plan Product line roadmap Platform economic scoping Platform requirements Product line/platform life cycle plan Investment and amortization plan IP licensing and royalties	Platform architecture specification Compatible IP/VCs (hardware, software, etc.) Interface specifications System-level models Platform characterization Standards references
Development and Integration Tools	**Support Practices**
Platform-based design and verification flow Application and system engineering support Software development tools and environments Application domain reference implementations and designs	Program management Integrator and platform/IP developer relationships Training and documentation Web presence Change/fix management Promotion and marketing

Table 4-1 Complete Platform-Based Development System (PBDS) Properties

Key to the PBDS are business plans and economic models built to analyze opportunities, costs, and returns from a platform and its derivative products. Investment in a platform, whether built or bought, must be amortized over all the derivatives to be built on it. This can result in significant cost savings and increased return on investment for a family of derivatives based on a common platform, where reuse is maximized, versus traditional "silo" products built one at a time with little or no reuse. Traditional single product business plans are inadequate for multi-product families.

Business plans must also consider platform evolution. Once a platform scope is defined and its architecture specified, it can evolve to support future generations of products (higher or lower performance, new technology implementation, additional or enhanced features, and so on). Effective platform evolution takes place in the context of the product family or derivative product set the platform supports. New generations of platforms that succeed or co-exist with prior generations may be planned in a product line roadmap, or evolution can be driven by unpredictable changes in markets, customer demand, or underlying technologies. But in either case, to have maximum benefit any platform architecture changes must be done with full understanding of the impact they will have on business, tool, technology, and support constituents of the PBDS.

PBDS development tools work in a product design flow that focuses more on integration and reuse of the platform, and related differentiating IP and VCs, and less on detailed design of all components. Selection, integration, configuration, and verification of products based on platforms and compatible IP is the primary challenge.

This chapter does not discuss all the attributes of the PBDS as shown in Table 4-1, but refers to elements of it throughout.

4.2 Platform Taxonomies

A taxonomy provides a means to categorize platforms according to a set of attributes. The attributes should be useful in distinguishing platforms intended for distinctly different purposes, including (if possible) different application domains and different levels of abstraction or implementation. The taxonomy establishes formal definitions that are concise and unambiguous for the various platform types. The taxonomy relies on the notions of orthogonality and separation of concerns, which represent an active thread in many research communities, conferences, magazines, and journals Descriptions and definitions for many of the terms used in this chapter are provided in Section 4.3, "Definitions."

The platform taxonomy represents attributes that are relevant to both platform designer-providers and platform consumer-integrators. Two sets of attributes are identified: those for the deliverable platform object, and those for the approach used to specify it. The Platform Object Complexity describes all the constituents of a deliverable "platform object" at a particular integration level as the combination of PBDS elements described in Table 4-1: its components and features, business and economic plans, development and integration environment, and support practices. The Platform Specification Approach describes types of platform specification processes. The platform development team will choose the most appropriate specification approach based on their business and technology philosophy, by which they will quickly converge on the desired platform specification and complexity.

4.2.1 Platform Object Complexity

4.2.1.1 Complexity Levels

We define Platform Object Complexity as a set of complexity levels, shown in Tables 4-2 through 4-4, covering stages of integration between simple IP blocks used in block-based design, and complete, physical SoC

devices ready for delivery and integration into a customer's higher-level product. Each complexity level describes deliverables in each of the major segments of the complete PBDS specification described earlier: components and features; business and economics; development and integration tools; and support practices. Each complexity level limits the number of attributes that are required to transfer between a platform provider and its integrator-user. A platform at a given complexity level can be considered as a virtual component by a platform integrator working at a higher level.

Complexity addresses topology and architecture, how the components making up a platform object are connected, and how they appear at the platform "surface," its external interface. As the number of components in a platform rises, the topological diversity and intricacy of the platform's organization increases. Each complexity level is a function of the integration effort, types of components, and deliverables. In our platform object taxonomy, a platform is specified by the complexity level of each of its major constituents, the hardware and software elements, business plans and models, integration tools, and support practices.

The lowest complexity level, "Set of Blocks," is not really a platform at all, but the result of traditional block-based design, where individual blocks or pieces of integratable IP are linked to create a usable device. ,The next level, "Core Platform," is a true platform at the sub-SoC level, integrating computing capability, some peripherals, and some software, not necessarily tailored for a particular application domain. Core platforms are used where the product specification process identifies an economic value at this lower level of integration, either as a component of a higher-level platform or of a customer's product. Core platforms may exist as a set of integratable soft models without any tangible implementation.

The highest complexity level, "SoC Platform,," is a complete, fully integrated and physically deliverable device, which may be in the form of a traditional single package, a flip-chip, multiple modules, or even a hard macro specifying fixed geometry and specific semiconductor technology. SoC platforms are defined when the product specification process identifies economic value at this level of integration. They may be derivatives of core platforms, by integrating domain specific functionality, design tool, and support with a lower level core platform.

Complexity Level: Set of Blocks A simple set of Virtual Components or functional IP blocks usable in traditional block-based design. (Not really a platform at all)			
Components/Features: HW Features: Processor and limited peripherals with point-to-point connections; no bus. SW Features: Few to none; may include register map, driver.	Business/Economics: Manufacturing oriented information only (for example, die-area costing in specific silicon technologies).	Development Tools: Only block-level verification. All hardware, software, and communication elements are provided along with block-based IP libraries.	Support: Block documentation, interconnection, and characterization data only. Components are not pre-clustered in any particular fashion.

Table 4-2 Platform Object Complexity Levels: Set of Blocks

Complexity: Core Platform A platform emphasizing computational capability, not necessarily application domain or market specific. For use in building higher-level SoCs (SoCs are core platform derivatives), or domain-specific products. Core platforms are defined at the required level of detail determined by the selected specification approach (see Section 4.2.2).			
Components/Features: HW Features (examples): 1) Single master processor or simple controller; single bus; (2) Single bus with multiple processor or controllers; (3) Multiple processors or controllers and/or multiple busses. SW Features: Platform initialization and startup code; RTOS kernel; basic on-chip peripheral drivers.	Business/Economics: Complete domain and economic scoping analysis showing economic impact of building and using the specific set of common features in the Core platform, including the on-chip variable features to be integrated with it to form higher level systems.	Development Tools: HW Support (examples): (1) Bus control timing and protocol; (2) Bus control timing and protocol, arbitration; (3) Bus definition, timing, and protocols, arbitration, synchronization. SW Support: Functional, behavioral, and verification models for the platform; compilers, debuggers, emulators, and so on.	Support: Not necessarily application domain-specific. Core platform architectural specification; platform-specific design support and reference implementations ; application engineering; training; change management; customer feedback.

Table 4-3 Platform Object Complexity Levels: Core Platform

Complexity: SoC Platform A packaged, physically deliverable device. SoC derivatives are higher-level systems produced by integrating the SoC with off-chip hardware and software. Includes all components and features of a Core Platform, plus domain-specific peripherals and/or differentiating HW/SW functionality.			
Components/Features: HW Features: Includes all specified hardware features (computing, memory, I/O, and peripherals, and so on); presented as packaged device, flip-chip, module set, or hard macro. SW Features: Includes all specified embedded software features (RTOS, drivers, APIs, external functions, and off-chip interface controllers programming models, and so on); presented as ROM/firmware or downloadable ROM image.	Business/Economics: Complete domain and economic scoping analysis showing economic impact of building and using the specific set of common features in the SoC platform, including the Core Platform and variable features integrated with it to form the SoC, and the off-chip variable features to be integrated with the SoC to form higher level systems.	Development Tools: Integration-oriented design flow including system models for the SoC, kernels, and clusters of appropriate IP components, including physical interconnect, I/O, timing, characterization, programming model, APIs, parameterization, verification, and so on.	Support: Likely targeted to a specific application domain or market segment. SoC platform architectural specification; domain-specific design support and reference designs; application engineering; training; change management; customer relations and feedback.

Table 4-4 Platform Object Complexity Levels: SoC Platform

Acknowledging the pragmatic mindset of many in the SoC community, we expect to see fully functional platforms offered at all complexity levels with fully specified component and feature attributes, but whose other PBDS constituents–business plans, development and integration tools, and support may not be as fully formed.

4.2.1.2 Interfaces

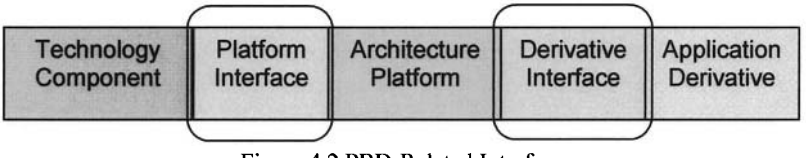


Figure 4.2 PBD-Related Interfaces

There are two basic interfaces in PBD, as shown in Figure 4.2: the platform interface between the traditional IC design and platform object creation, and the derivative interface between the platform object and

derivative creation. Depending on the specific methodology, each of these interfaces has a wide range of variation. (Note that while we have shown two interfaces in this example at the SoC level, an end product may have many such layers between integration platforms. Further research may define more such layers in the SoC space.)

Platform Interface

This has as little as a traditional ASIC vendor offering with complex IP libraries, to as much as reference designs and frameworks for compatible IP. Finished designs or platforms can be constructed from the information, which crosses this interface.

Derivative Interface

This has as little as a basic platform model, along with the information available at the platform interface, to as much as a complete platform with well defined hardware, software and application interfaces from which derivative designs can be created.

In the technology-driven case the vendor supplies all the components, ideally in an easily integratable form. In this case, the designer sees only the platform interface. The vendor deals only with the designer through the platform interface. The designer has the components from which to build a platform or a finished design, but must integrate and verify the design in the traditional (such as, ASIC) manner.

In the architecture-driven case the derivative designer sees both the platform interface and the derivative interface, since the platform is being supplied by one vendor and the technology is being supplied by another. In the ideal case the components are compatible with the platform and the implementation path has been verified. But the derivative designers still must define their application interfaces, integrate and verify their design in a traditional ASIC manner, ideally with less effort given the level of integration and verification framework that comes with the platform.

In the application-driven case the derivative interface has all of the interfaces necessary to easily tailor the platform to meet the derivative design requirements. In this case the platform interface is hidden from the derivative designer. The effort to integrate and verify the final derivative design should be limited to plugging the custom components into the existing framework and environment and verifying them at the application interface. In the ideal case the platform provider can also easily create the platform and all the necessary interfaces for the derivative designer from the components and tools available to him. In this case the derivative designer specifies the platform and then creates the derivative design. This latter case

is quite likely even if the platform is custom-made for the derivative designer because the derivative designer only cares about his differentiating IP or variable functionality, not the common features and components of the platform.

Detailed specifications for the interface requirements can be created, but the items beyond the traditional block-based design requirements are largely conditional, based on both the type of methodology, and the underlying technology. For example, an application-driven approach for an FPGA-based platform may require no hardware development, while a technology-driven approach for a standard cell-based platform may require considerable hardware development.

4.2.2 Platform Specification Approaches

We have identified three basic approaches to platform specification: technology-driven (bottom-up); architecture-driven (middle-out); and application-driven (top-down). Product family planners and platform architects will use one of these approaches to specify their desired platform and its integration environment, within the context of their business goals, overall product family strategy, technology capability, and other constraints. The product of the specification process will be a platform at an appropriate level of complexity, as defined in Section 4.2.1. The remainder of this section provides a short description of each approach, followed by a detailed listing of distinguishing attributes we have identified.

4.2.2.1 Technology-Driven (Bottom-Up) Specification

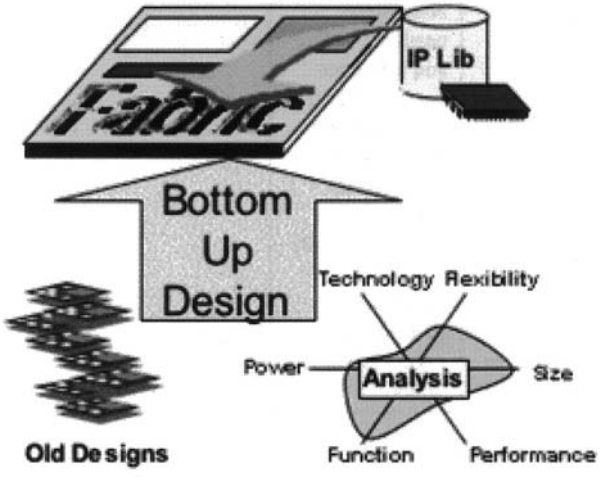


Figure 4.3 Technology-Driven Platform Specification

Technology-driven platforms use a bottom-up approach to platform specification, based on traditional design methods (standard cell, full custom, library-oriented, and so on) with the latest and potentially highest performance (and costliest) semiconductor technology processes. They are agnostic with respect to applications, though applications with higher performance and/or integration needs will likely be the first users of newest technology-driven platforms (including 130 nm node, 90 nm node and predictions for future nodes such as 65 and 45 nm.). Examples include Intel processors and platform FPGAs wanting to offer much greater integration.

At this level, all the hardware, software and communication elements are provided along with the block-based IP libraries. Architecture reference models are provided but the components are not pre-clustered in any particular fashion. This is viewed as a bottom-up platform methodology in that specific derivatives are created up from the component level using the architectures provided as a reference only.

4.2.2.2 Architecture-Driven (Middle-Out) Specification

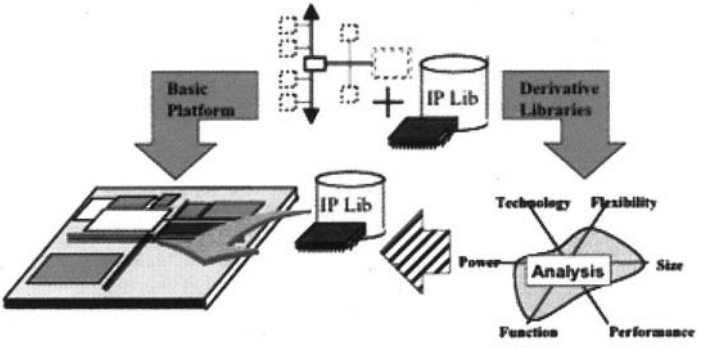


Figure 4.4 Architecture-Driven Platform Specification

Architecture-driven platforms use a middle-out approach to platform specification; a system level approach with restrictions derived from the relevant technology foundation (the family of existing cores, core support packages, memory structures, and on-chip communications standards). They have loose coupling to both applications and technologies, primarily driven by new architectural paradigms, though provided in consideration of both technology (such as, power, size and speed) and applications (such as, RTOS support and memory hierarchies).

At this level, specific kernels including diagnostic shells, RTOS software, processors, and communication structures are included in a pre-verified, architecture. These architectures have attributes that are targeted to specific market segments. System models for these kernels and clusters of appropriate IP components are identified along with methods for integrating them with the kernels into specific derivative designs. This is viewed as a middle-out platform methodology in that a pre-defined kernel containing the basic architecture already exists, and is used along with pre-verified component IP to create a derivative.

4.2.2.3 Application-Driven (Top-Down) Specification

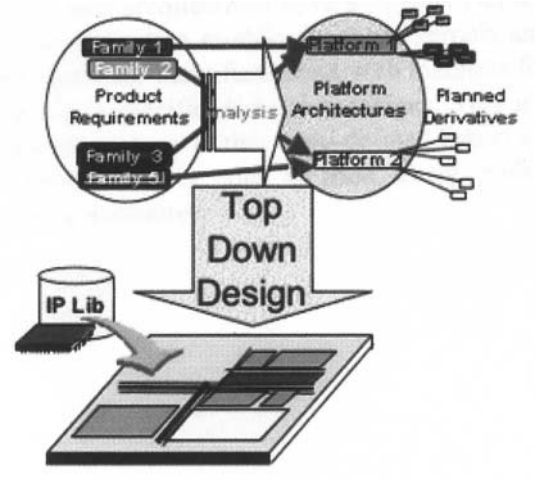


Figure 4.5 Application-Driven Platform Specification

Application-driven platforms use a top-down approach to platform specification, using system-level design methods and focusing on the functional requirements of family of products (set of derivatives) to be built on the platform. At this level there is a top-down, product family roadmap-driven process for creating a platform and the derivative products based upon it. They are agnostic with respect to the fundamental software and semiconductor technologies used to build the SoC, though the underlying technology must offer suitable application "performance plateaus," that is, enough performance to meet application requirements.

The development team uses the roadmap to drive creation of an appropriate platform (the set of common IP or features), including application interfaces, and application oriented implementation /verification flows, from the component libraries, platform frameworks, meta-methods and meta-applications interfaces. Product developers integrate the platform with other IP (the differentiating or variable features) to produce the desired platform derivatives, the product family members.

4.2.2.4 Platform Specification Attributes

Attributes of the three platform specification approaches identified in the PBD taxonomy are summarized in Tables 4-5 through 4-11, and platform development teams should use these attributes to determine which specification approach is most appropriate for them.

Attribute	Technology Driven	Architecture Driven	Application Driven
Approach	**Bottom-Up**	**Middle-Out**	**Top-Down**
Platform architecture.	Implied, but not fully specified, by library components. Incorporation of embedded processors brings a lot of architecture "baggage" (on-chip communications, memories, some peripherals, basic SW development tools, and so on) that begin to "nucleate" a specific architecture.	Implied by the technology foundation (TF), though minimal implied restrictions to remainder of system. Should allow optimal integration in market-targeted platform.	Explicitly built into the platform, though can be parameterized. The user (integrator) may not be concerned with what is inside the platform's API and/or standard bus boundary. Need sufficient observability and controllability for derivative product validation.

Table 4-5 Platform Specification Approach Attributes: Architecture

Attribute	Technology Driven	Architecture Driven	Application Driven
Approach	**Bottom-up**	**Middle-out**	**Top-down**
Platform-based product roadmap.	May be *ad hoc* or just for initial product based on the platform. May be based on a general technology roadmap, that is, process geometry node or number of gates per die expected.	Depends upon flexibility and scalability of the TF. Roadmap will be defined both by target markets (such as, wireless), as well as suitability of the TF to support current architectural styles (such as, most dominant OS in market, and so on).	Well-defined product family roadmap precedes platform design. Roadmap should be well thought out over medium to long term (2-3+ years), with many derivative products based on the platform, and planned platform evolution.
Derivative products.	Only first application, not a complete set of derivatives, may be well known. First application may be known and is platform driver. Derivatives could be application platforms; may be product-level sequential entities	A "market-targeted platform" or an application-specific platform (ASP) can be derivatives of a TF. A TF can form a kernel in both a bottom-up or top-down flow. Having a defined TF makes is possible to work out how many different variants can be added to it.	Initial family of derivatives well defined (part of the overall product line strategy). Multiple derivatives can be created simultaneously and in parallel from a single defined platform.
Marketability as a stand-alone "platform product."	High. Public offering of a platform product easier, as it's not tied to specific application. Compare more generic platform offerings.	High. The more simply defined the boundaries of the TF is, the more "add-ons" can be integrated, and the easier it is to market. A TF may be defined as: 'the absolute minimum HW support required for OS porting on a core, providing a standard OS driver-API and bus interface'. This implies that memory	Low. Platform design and capabilities are strongly tied to the defined product line (set of derivatives). The platform is likely to be kept as an internal use entity, or shared with 'favorite' customers, such as, TI OMAP with Nokia and others.

Attribute	Technology Driven	Architecture Driven	Application Driven
		maps, and so on are not set, providing greater freedom to the customer. The more flexibility provided for interfacing to the TF, the simpler to adopt. (See Note 1.)	"Internal" means partners within the entire product chain (to end user), whether inside or outside the SOC provider's company.

Table 4-6 Platform Specification Approach Attributes: Market

Attribute	Technology Driven	Architecture Driven	Application Driven
Approach	**Bottom-up**	**Middle-out**	**Top-down**
Platform designer's driving question.	"What can I put in the platform that's cool?" Statistically, how can I provide the greatest number of customers with what they need? "What customers need" is not totally dissimilar to the top-down approach (that is, the common features of the product family), but tends to be raw feature-oriented, independent of well thought out applications. Also, "How can I fill my new fab most effectively?"	"What HW/SW support do I require to facilitate our TF?"	"What do I need to put in the platform to support my product line?" Platform contains just enough to support product family, and no more. Economic scoping models and management of feature sets are on a par with traditional "engineering" point of view.
Platform is agnostic with respect to.....	Applications. Applications with higher performance and/or integration needs will likely be the first users of newest technology-driven platforms, such as, 130 nm node and predictions for future. Example: Intel processors or	Loose coupling to both applications, technologies. Primarily driven by new architectural paradigms, though provided in consideration of both technology (such as, power, size, and speed) and applications (such as,	Technologies. Underlying technology must offer suitable application "performance plateaus," that is, enough performance to meet application requirements. (See

Attribute	Technology Driven	Architecture Driven	Application Driven
	Platform FPGAs wanting to offer much greater integration.	RTOS support and memory hierarchies).	Note 2.)
Creation emphasis at ...	Component and interconnect levels.	Technology foundation exploration; capacity to replace TF and evaluate performance at a level with sufficient accuracy.	System and product levels, plus middle-out IP integration.

Table 4-7 Platform Specification Approach Attributes: Creation

Attribute	Technology Driven	Architecture Driven	Application Driven
Approach	**Bottom-up**	**Middle-out**	**Top-down**
Platform stability. (See "platform change trigger.")	Volatile. New platforms developed as new technology becomes available.	Stable. High stability, based on the technology foundation.	Stable. Platform enhancements or new platforms developed as part of overall product line roadmap.
Platform change trigger.	Capability breakthrough or planned evolution in key technology. Example: Change from 130nm to 90nm process node.	Technology foundation (TF) change. Example: Change from 32-bit to 64-bit computing cores.	Product line need, driven by feature set evolution, or new standards. Example: 2G wireless to 2.5G with GPRS; 2.5G to 3G with UMTS)

Table 4-8 Platform Specification Approach Attributes: Alterations

Attribute	**Technology Driven**	**Architecture Driven**	**Application Driven**
Approach	**Bottom-up**	**Middle-out**	**Top-down**
Platform creator's design method.	Traditional design (standard cell, full custom, library-oriented, and so on).	System level approach with restrictions derived from the technology foundation.	System-level design.
Platform customer/ integrator's design/ development method	Traditional, seeking optimization. Integrator may want to tweak platform internals to optimize the product into which it is going. (See Note 3.)	Reuse without rework of the TF. Strong focus on optimization of HW/SW components around the TF; TF must be provided with a configurable and extensible verification and validation (V&V) harness. Ability to configure hardware subsystem provided external to TF. Ability to easily integrate IP from compatible libraries (HW and ESW). TF must be provided with an OS port.	Reuse without rework of the SOC or core platform. Strong emphasis on product-level integration using the platform. The platform itself may be configurable at run time or design time, but not changeable. Platform must be provided with a configurable and extensible V&V harness.
Derivative methodology.	Traditional library oriented design; typically hardware with some software.	HW/SW architectural exploration and IP integration.	Application specific interfaces for system augmentation; typically software with some hardware. Set of selectable IP common to the market-target of the platform likely to be provided as a library for derivative product design/assembly.

Attribute	Technology Driven	Architecture Driven	Application Driven
Verification and validation (V&V) support for platform-based products.	Traditional bottom-up design verification of entire platform-based design.	Reusable, configurable, and extendable verification IP harness. Verification IP communicate through standard interface. SoC structure, such as memory-map, inherited by V&V environment from a single source.	Verification IP harness and system-level verification.

Table 4-9 Platform Specification Approach Attributes: Creation Methodology

Attribute	Technology Driven	Architecture Driven	Application Driven
Approach	**Bottom-up**	**Middle-out**	**Top-down**
Platform delivery form (hard-firm-soft). NOTE: SoC-level platforms will be delivered as die, modules, packaged devices, or hard macros.	Chip, hard or soft. May be provided hard with configurable sections.	Chip or soft.	Chip or soft. Most likely provided soft with a reference methodology adaptable to different technologies.
Degree to which information is hidden; "black boxiness." (See Note 4.)	Low. User may want lots of visibility to platform internals – may want to optimize implementation for performance and integration. (See Note 2).	Medium. Degree to which information is partially visible; "gray boxiness" (that is, V&V support needed).	High. User may not care what's inside the platform boundary. As long as useful and effective V&V controllability is available for SOC platform, derivative product TTM concerns are paramount and drive reuse without rework.

Attribute	Technology Driven	Architecture Driven	Application Driven
Real-World Examples of this Platform Type	Xilinx Virtex II Pro	ARM Integrator ARM PrimeXsys™	OMAP (TI) Nexperia (Philips) Innovative Convergence (Motorola)

Table 4-10 Platform Specification Approach Attributes: Delivery

Attribute	Technology Driven	Architecture Driven	Application Driven
Approach	**Bottom-up**	**Middle-out**	**Top-down**
Available software components.	Hardware abstraction layer (HAL). Example: Drivers.	Hardware-dependent Software (HdS). Example: Drivers, HdS-compliant API, OS abstraction layer, and so on.	Application software. Example: MPEG-4, MP3, OS abstraction layer, and so on.
Platform customer/ integrator's involvement in platform development.	Loosely coupled; feature-based, performance specification oriented. Closer to a "standard product."	May be involved in the development, though not compulsory. Depends upon whether foundation IP is developed in a "collaborative" specification environment.	Strongly coupled; top-to-bottom architecture consistency with customer's application. Closer to a "custom product." Example: Ericsson and Nokia involvement with TI for wireless.
Customer. ("Levels" means all integration levels: for SoCs.)	Internal or external at all levels. May be at least two levels (core/sub-SOC and SOC boundary), or just one (SOC boundary only)	Internal to a systems-company, or external. TF fundamentally ensures support for efficient integration of complex IP objects.	Internal (vertically integrated companies); selected external partners. Used by partners within the entire platform-based product food chain.

Table 4-11 Platform Specification Approach Attributes: Customer Support

Notes to Tables 4-5 through 4-11:

1. A good analogy with the architecture-driven or technology foundation (TF) approach is the PC, which has a well defined TF, but also an ample degree of flexibility. For example, if one wanted a PC to be a video-conferencing system, one must add the hardware (webcam, modem, and so on) and the software necessary to customize it. Different TFs (such as Pentium 4, Pentium 3, Athlon, Celeron, Power PC; Linux, Windows and Mac OS X) have different degrees of freedom and are more or less restrictive when adding peripheral hardware and software. Having a defined TF, one can work out how many different variants can be added to it.
2. Arguably, SOC platforms for 2G wireless standards were not possible until process nodes around 250nm to 350nm offered enough performance-power tradeoff and integration, while 3G demands a 180nm to 130nm node.
3. Modifying the internal details of a platform tends to invalidate the concept. For example, a technology-driven platform consisting of cell libraries and generators usually does not allow modification of the basic cell libraries; rather, new cells may be added for specific design needs. Thus the basic cell libraries may represent the invariant or configurable-only part of a technology-driven platform. An essential characteristic of the platform is isolation between application development and platform implementation detail.
4. Configuring or customizing of a platform through a specified design-time or run-time parameterization or other configuration process is acceptable, but user manipulation of the platform internals (for example, in search of "optimization") is not. Thus a highly configurable processor (such as Tensilica T1050 or LX) may be part of a platform post-configuration, but the process of optimizing a processor configuration for a specific application lies outside of the processes of platform-based derivative design, and instead can be part of the platform creation process.

4.2.2.5 Metrics for Platform Specification Approaches

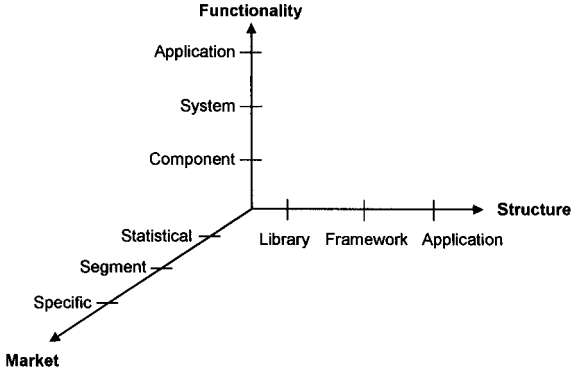


Figure 4.6 The Platform Metric

There is a relationship between the level of detail needed for a set of platform-based products, as measured in three metrics: functional abstraction, structural creation, and market alignment, and the approach that should be used to specify the platform. Figure 4.6 shows successive levels of detail from none (at the origin) up to highly detailed levels of organization on each metric. The remainder of this section gives a short description of each metric and its relationship to the different specification approaches.

Functionality

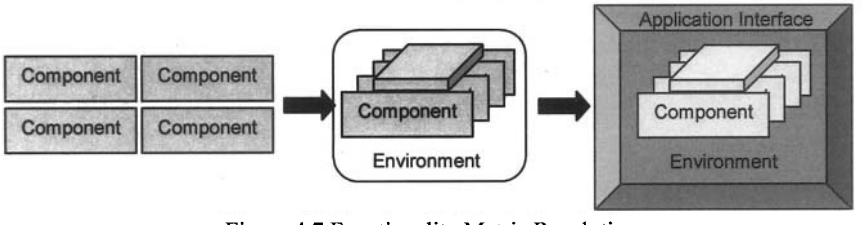


Figure 4.7 Functionality Metric Resolution

The lowest level of functional detail deals with simple, unmanaged collections of components: processors, memory, I/O translators, and compute engines. These may be combined into complex aggregations, but the functionality focus remains primarily with the performance of the individual components and limited, non-application specific interactions between them. Structural and high-level models are provided for the components. At intermediate level of functional detail are descriptions of

components and the communication between them as the basis for a system, and the beginning of functionality management. Mid-level models exist, complete with sufficient diagnostics, and interfaces to perform system level functional verification, with strong linkage to component properties and some linkage to applications. The highest level of functional detail deals with fully managed sets of features and interfaces for the target applications themselves, driven by the platform user's interests and concerns. This can include very high-level models, coupled with interfaces and examples that ease the application development.

Structure

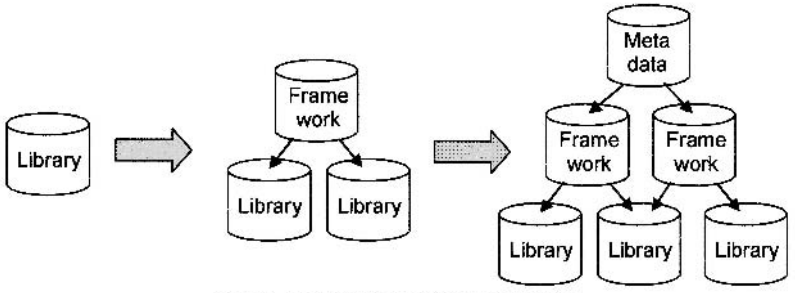


Figure 4.8 Structure Metric Resolution

At the lowest level of structural detail, the platform consists of a library of interconnectable, reusable components. The library will likely include market, architectural, and system-level information, but is still organized by components. At the next level the structural detail also includes frameworks for the common components or features that will cross multiple components. The framework includes platform-specific information, both with respect to the common components, and kernel architectures for the platforms. Some of this information includes verification suites, diagnostic shells, and parameters for general implementation tool suites. At the top level the structural detail includes both component libraries and platform frameworks, but has an additional layer of meta-data such as product family-level meta-applications interfaces, attributes, and meta-flows used in specifying and constructing platforms.

Market

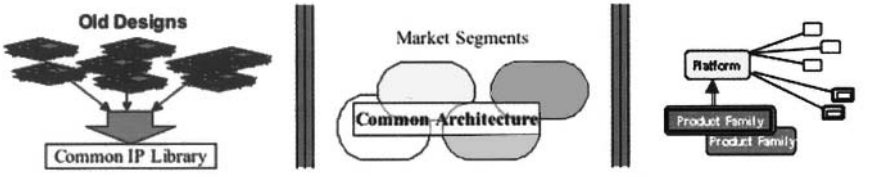


Figure 4.9 Market Metric Resolution

At the lowest level the market detail and derivative product mix is not well defined; it is merely a statistical composite of the market for the platform. At the next level the market segments are identified, as are the common elements within the segments. At the highest level a specific set of derivatives of the platform are identified, and the common elements of their specifications defines the platform.

4.2.2.6 Alignment with Platform Specification Approaches

The different approaches to platform specification can be viewed as concentric shells defined by the level of detail in the platform space whose dimensions are functionality, specification, and markets. In Figure 4.10, we see how these metrics for a set of axes create a "platform approach space." We recommend that every product development program consider its market, functionality, and structural detail, then adopt the platform specification process defined by the shell with which it most closely aligns.

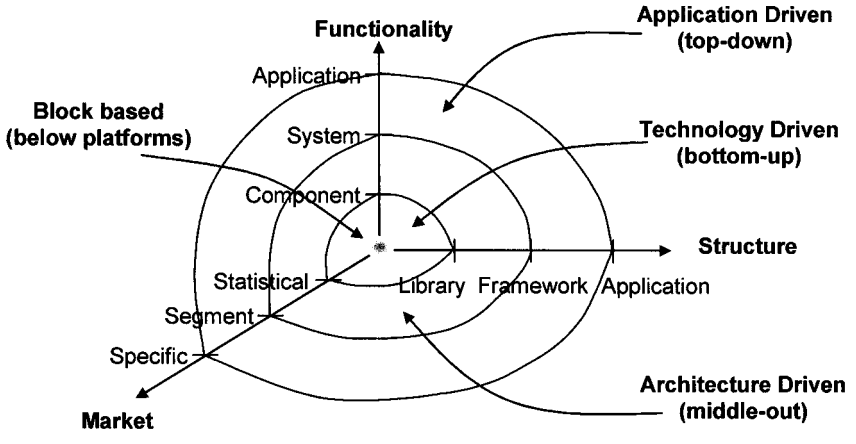


Figure 4.10 Platform Specification Approach Space

Block-Based (No Platform)

This level assumes the availability of a reusable IP library, and existing general tool flows for implementation around a block based methodology. No architecture or platform is implied at this level.

Technology-Driven Approach

At this level all the hardware, software and communication elements are provided along with block-based IP libraries. Architectural reference models are provided but the components are not pre-clustered in any particular fashion, and with little detail. We view this as a "bottom-up" platform specification approach in that specific derivatives are created up from the component level using the architectures provided as a reference only.

Technology Foundation/Architecture-Driven Approach

At this level specific kernels including diagnostic shells, RTOS software, processors, and communication structures are included in a pre-verified, architecture. These architectures have attributes that are targeted to specific market segments, and moderate levels of detail in all metrics. System models for these kernels and clusters of appropriate IP components are identified along with indications of how they can be integrated with the kernels into specific derivative designs. This is viewed as a middle-out platform methodology in that a pre-defined kernel containing the basic architecture already exists, and is used along with pre-verified component IP to create a derivative.

Application-Driven Approach

At this level there is a top-down, product family roadmap-driven process for creating a platform and the derivative products based upon it, and high levels of detail in each metric. The development team uses the roadmap to drive creation of an appropriate platform (the set of common IP or features), including application interfaces, and application oriented implementation and verification flows, from the component libraries, platform frameworks, meta-methods and meta-applications interfaces. Product developers integrate the platform with other IP (the differentiating or variable features) to produce the desired platform derivatives, the product family members.

4.2.2.7 On The Evolution of Platform Specifications

While the three concepts of technology-driven, architecture-driven, and application-driven platforms are distinct approaches to platform specification, the concentric, ever larger shells imply the possibility of a progression. The rest of this section will discuss the requirements for moving from one level to the next. Figure 4.11 shows how a platform specification at the segment-system-framework level (architecture-driven) in the platform space might evolve to the segment-application-application (application-driven) level.

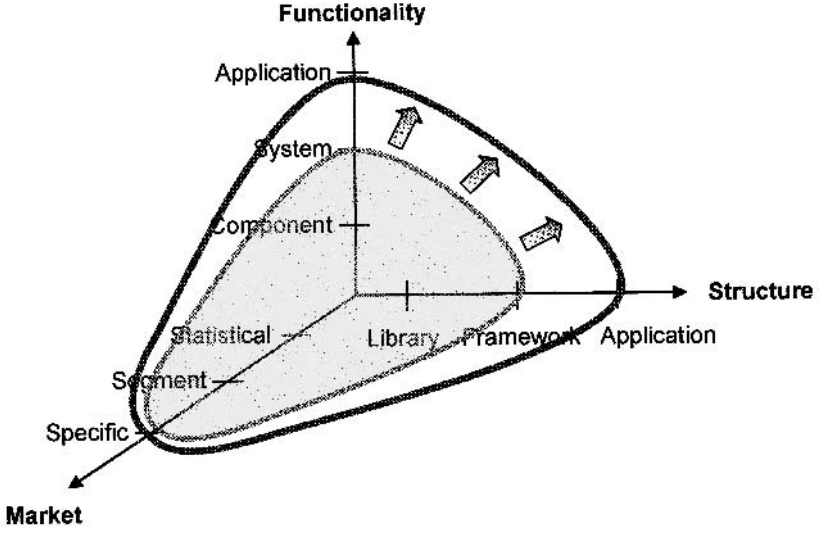


Figure 4.11 Evolution in Platform Space

Block-based design assumes a library of qualified reusable IP and general design flows that support the integration of reusable IP into a wide variety of end user designs.

Evolution from block-based design toward technology-driven platform specification is the process of providing clusters or groups of IP that have been further verified to work together in a wide variety of combinations, and for which the existing reusable IP libraries have been extended, to include software IP, RTOS modules, diagnostics, and reference architectures for the clusters. High-level component models are typically provided at this level as well, but full system models as references are not generally provided.

As a platform specification approach evolves from technology-driven towards architecture-driven, the level of detail increases and kernel architectures complete with system models are added to a second level of the library. The component IP will have distinct and defined relationships to the kernel architectures added to their other attributes in the library. Integrated diagnostic and software application strategies are identified and market segment descriptions for each of the platforms are added to the database. At another level, the technology-driven, a basic IP library has additional information while, at the architectural-driven level, the database is more platform-centric. Specific options for the general tool flows and implementation are tied to the specific platforms. Typically a technology-driven platform provides all available options to the component IP, which are limited to viable options within each specific architecture at the architecture-driven level.

Finally, moving from architectural-driven to application-driven platform specification involves considerable detail about the functionality, market, and structure of the product family using the platform, and the addition of top down methodologies for creation of platforms, along with application interfaces and preconfigured application oriented implementation and verification cockpits. These interfaces presume full very high-level system modeling at the application level, which in turn is calibrated to specific architectural system level models. Of course, in order to efficiently create top down platforms, it is necessary to create meta-application interfaces, and configurable implementation and verification cockpits that are easily configured for specific applications. These application-oriented interfaces allow the creation of derivatives from the application perspective without the need for detailed knowledge of the underlying architecture. In short, this layer views the platform as a solution to a specific product roadmap, as opposed to general market segments, and the data base of a completed platform would reflect the proposed roadmap as part of the market data with respect to the platform

4.3 Definitions

Platform.	An integrated and managed set of common features, upon which a set of products or product family can be built. A platform is a virtual component (VC). From Meyer & Lehnerd [MEY 97]: A set of subsystems and interfaces that form a common structure from which a stream of derivative products can be efficiently developed and produced.
Platform-Based Design	An integration oriented design approach emphasizing systematic reuse, for developing complex products based upon platforms and compatible hardware and software VCs, intended to reduce development risks, costs, and time to market.
Integration Platform.	In the SoC context, a library of virtual components and an architectural framework, consisting of a set of integrated and pre-qualified software and hardware virtual components (VCs), models, EDA and software tools, libraries and methodology, to support rapid product development through architectural exploration, integration and verification. Within the VSIA, the scope of the integration platform is bounded to the SoC. From Chang, et. al. [CHA 99]: Hardware architecture, embedded software architecture, design methodologies (authoring and integration) design guidelines and modeling standards, VC characterization and support, and design verification (hardware/software, hardware prototype), focusing on a particular target application.

Table 4-12 Basic Definitions

Platform Provider.	An organization, company or individual that offers and supports platforms for others to use in their designs.
Platform Taxonomy.	The taxonomy is defined by a series of matrices containing levels of platform types, characteristics of platforms, application areas of platforms, platform tools, design styles and other properties necessary to describe the platforms.
Virtual Components	With respect to platforms, virtual components are functional blocks of hardware or software that can be assembled to create a platform, platforms themselves, or differentiating IP features or blocks integrated with a platform to form a higher-level platform or product. In other contexts, virtual components may be thought of as just components.
Technology Foundation	The family of cores, core support packages (local HW support for OS ports on a core), memory structures, and on-chip communication standards used in platform assembly and platform-based design flows.
Hierarchy	An ordered list of platforms, for an application or a view, in which the lower-level platforms are the sub-platforms that make up the higher-level platform. The lowest level of the hierarchy is a virtual component (or component)
Platform Levels.	An implied level of abstraction, such as a higher-level platform, is one that contains lower-level platforms and/or virtual components. While platform-based design is applicable at many levels of integration within a finished product, our scope will be limited to platform-based products at the SoC level and below. SoCs, which themselves may be built on lower level platforms, will serve as platforms for higher-level products.

Platform Object.	A deliverable platform at some level of abstraction, comprised of an integrated, defined set of components and features, and business plans, development tools, and support practices.
Platform-Based Development System. (PBDS)	The complete development environment that enables a platform customer-user to integrate a platform and variable IP into a higher-level derivative product. PBDS constituents are components and features, business plans, development tools, and support practices.
Platform Complexity Levels.	Describe all the constituents of a deliverable platform object at a particular integration level as the combination of PBDS elements: its components and features, business and economic plans, development and integration environment, and support practices.
Platform Specification Approach.	A type of platform specification process chosen by the platform development team, based on their business and technology philosophy, by which they will quickly converge on the desired platform specification and complexity.
Platform Evolution.	The notion that a particular type of platform is improved over time by extension (such as, adding functionality, incorporating additional modules or platforms, enhancing performance, and so on), or by migration (such as, changing the basic computing complex, being manufactured with new semiconductor technology, and so on). Successive generations of platforms may be defined as part of a product family strategy; each platform is fully verified before release for integration into derivative products. Because of isolation between the platform internals and its supported interfaces, platforms can evolve without obsoleting applications that use them. The target application domain typically defines the platform type, which remains the same throughout the evolution of the platform.
Derivative	A specific instance of a product or design based on a particular platform. One member of a product family based on a particular platform.
Isolation	The distinction between the internal features and construction of a platform and its external characteristics, as known from its interface, which permits the internals of a platform to evolve while keeping consistent the external view seen by the platform integrator.
Model	A platform model is an abstraction or several abstractions of the platform that can be used by hardware, firmware, and software developers to create products. The model is a black box that exposes the functionality that is necessary for the developer to do his/her job. The EDA tools and software tools needed to use the model must be delivered with the model by the platform provider.
Viewpoint	A pattern or template from which one can develop individual views by establishing purposes and audience for a view and the techniques for its creation and analysis. A viewpoint specifies conventions for constructing a view.
View	A representation of a whole system from the perspective of a related set of concerns.
Formal Platform Architectural Descriptions	Architectural descriptions contain the information about the platform from the perspective of all the stakeholders, identifying all their concerns, and addressing those concerns via architectural viewpoints. Architectural descriptions are defined by the functional and collateral virtual components of platforms.
Architecture	A collection of architectural descriptions from which describes the platform.

Platform	
API Platform	A layer of abstraction of a complex device or system that provides the basis the design process. This typically is a programmer's model of the system or device.

Table 4-13 Higher-level Definitions

Product Family (Product Line)	A group of products sharing a common, managed set of features that satisfy specific needs of a selected market or mission area, and that are developed from a common platform or set of essential assets. From Meyer & Lehnerd [MEY 97]: Individual products that share common technology and address related market applications.
Domain	An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.
Domain Analysis	A process for capturing and representing information about applications in a domain, specifically identifying common characteristics and reasons for variability.
Economies of Scope	The condition wherein fewer inputs such as effort and time are needed to produce a greater variety of outputs. Economies of scope occur when it is less costly to combine two or more products in one production system (that is, to build them as part of a platform-based product family) than to produce them separately.
Essential Asset	An artifact that is used in production of more than one product in a product line. A core asset may be an architecture, a hardware or software component, a process model, a plan, a document, or any other useful result of building a system.
Product Family Architecture	A description of the structural properties for building a group of related systems (that is, a product family), typically the components and their interrelationships. The inherent guidelines about the use of components must capture the means for handling required variability among the systems (sometimes called a reference architecture).
Product Family Approach	A system of production that uses assets to modify, assemble, instantiate, or generate a line of products.
Common Features	Features that are shared all members of a product family, and can be integrated into a platform. The extent of common features is determined by domain analysis and the economy of scope for a given product family (see integrated and managed features).
Variable Features	Features that differ between product family members based on a common platform (also known as variable IP or differentiating IP.)
Integrated and Managed Features	A set of features that have been selected and configured via a defined domain analysis and economic scoping process (managed), then integrated, tested, verified, and validated (integrated) to form a reusable virtual component.
Architecture	The preferred definition, from IEEE-1471 : The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution. Examples of other definitions:

	From Clements & Northrop [CLE 02] : The structure or structures of a system, which consists of hardware and software components, the externally visible properties of those components, and the relationships among them. From Section 2.9.2.3: • In the context of software, architecture is the configuration of all software routines and services for meeting a system's objective. For example, application, operating system, and communication protocols can describe layers of a software architecture. • In the context of hardware, architecture is the configuration of all physical elements for meeting a system's objective. • In the context of systems, architecture is the collection and relationship of the system's constituent hardware and software components. For example, a multiprocessor system's architecture would include the hardware network architecture and the software architecture in the form of distributed and local operating systems, and application and control routines.
Functional-to-Physical Architecture Mapping	A "mapping" between a functional model of a system and a specific hardware and/or software implementation architecture that implements the functions implied by the functional model (a functional model will have an architecture of its own, independent of the implementation). Such a mapping explicitly ties together the main structural components of the system, in both the hardware and software senses, and the functions that are associated with each of them. This mapping should deal with the required communications between functions, and the components in the architecture that are used to realize this communication (for example, in the hardware domain, buses; in the software domain, messaging routines; between hardware and software, memory-mapped I/O). In some contexts, the combination of the architecture of a system, the functional model of the system, and the mapping between the two, along with associated implementation constraints, is called an "architecture."

Table 4-14 Product Family Definitions

CHAPTER 5 HARDWARE-DEPENDENT SOFTWARE

5.1 Introduction

This chapter expands the scope of models and definitions into the world of embedded systems. Recognizing the increasing complexity of modern SoCs, which incorporate both hardware and software IP, the productivity gap is not solvable by only looking at hardware IP portability. Issues such as exposing SoC functionality using a software API; the portability of software running on evolving hardware platforms; and the increasing issue of SoC verification and validation, lead to a requirement to clearly define the interfaces between a hardware platform and any software layers that run applications using that hardware platform. The software layers that use these interfaces are collectively known as "hardware-dependent software" (HdS).

This chapter explores the following objectives:

- To what extent can a dedicated software layer be defined so that it efficiently hides hardware platform specifics from application software?
- How can we enable software portability across various hardware platforms?
- How can we allow cost-effective integration of software IP from different vendors into overall solutions?

5.1.1 Purpose of this Chapter

- To fix the definition of terms that are important in the HdS context
- To clarify the subject, considering its different aspects, such as hardware and software platform views, and its relation to different HdS design-cycle phases

- To classify the relationship and interactions between hardware and software
- To help hardware, EDA, and software engineers understand the terminology of other experts in relation to HdS

5.1.2 Intended Audience

Since HdS constitutes the border zone between hardware blocks (ASICs, FPGAs, processors, printed circuit boards, and peripherals) and the functions implemented in software which sit on top of these hardware platforms, there are several different users who will find this chapter useful:

- HdS designers and engineers: This chapter offers a vocabulary in the HdS domain and provides a means of unambiguous communication. It facilitates the definition of other related topics, such as the HdS Application Programmers Interface (API). (See Section 5.3.5.4.)
- Hardware designers and software engineers: For them, this chapter is mainly a tool for understanding the specifics of HdS, and to make efficient use of the HdS concept in their respective domains.
- System architects, integrators, and testers: As primary users of the HdS API, they need to understand the different aspects (life cycle, hardware platform, and runtime) of the HdS concept.
- EDA/IP Developers: As tool and IP developers explore and automate functions for both hardware and software design and development, this chapter provides a structure for common understanding across the users and the developers of these automated functions.

The introduction and correct understanding of the HdS concept assists the following:

- SoC providers that offer highly portable IP as virtual components.
- RTOS and software companies that support a growing variety of hardware platforms with a reasonable development effort
- System houses that efficiently integrate complex products by incorporating best-in-class IP into working solutions, because they can concentrate on functional rather than implementation aspects.

5.2 HdS Terms and Abbreviations

This section gives definitions for a basic HdS-related terminology. First, it defines the meaning and the purpose of the two basic concepts of hardware-dependent software and of the hardware abstraction layer (HAL).

Then, it gives a list of abbreviations used in the HdS context. It concludes with a list of HdS-related terms with their meaning in the context of this subject.

5.2.1 Basic HdS Definitions

For a list of common acronyms, the reader should consult the "Abbreviations and Acronyms" section at the front of this book.

5.2.1.1 HdS (Hardware-dependent Software)

All software that *directly depends* on the underlying hardware belongs to HdS. The following software items are examples:

- Hardware drivers and the HAL
- Boot strategy, and boot loader
- Built-in tests (basic level, offline tests, system maintenance tests)
- Hardware-dependent parts of communication stacks
- Algorithms implemented in software on DSPs

HdS shields the hardware from upper-layer application software. Communication with the hardware takes place using a stable API, the HdS API. HdS contains all software that is needed to validate, verify and test, and to bring up the underlying hardware platform. HdS retains portability across various simulation and target environments.

Figure 5.1 illustrates the relationship of the hardware-dependent software to the hardware design, software development, and manufacturing activities required to produce a product. Furthermore, the bridge between the hardware and software is the software layer closest to the hardware, the Hardware-dependent Software (HdS). On this abstraction level, the HdS is the border zone specifying the platform-dependent mapping between hardware and software resources.

From a test strategy perspective for both development and manufacturing, the HdS architecture must be set up as a basic general framework of the test software and platform. The HdS architecture needs to include methods for test initiation, parameter-passing, and the processing and reporting of anomalies and exceptions.

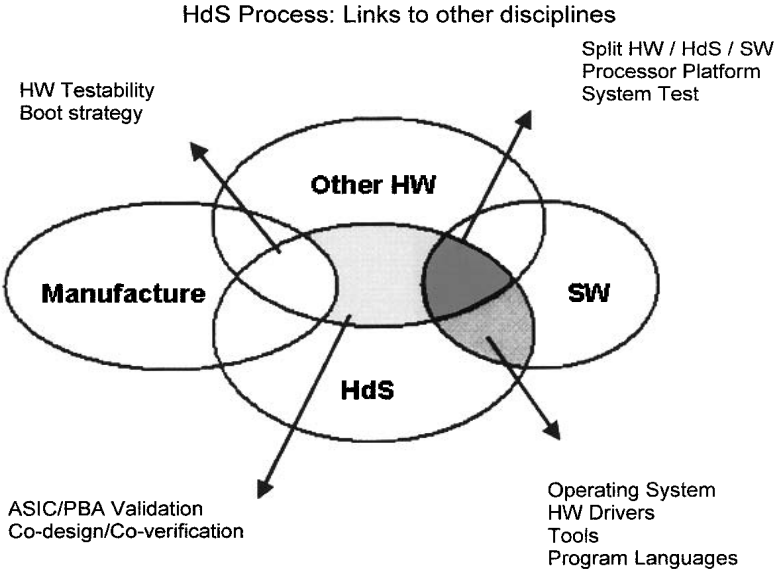


Figure 5.1 The Relationship of HdS to Design, Development, and Manufacturing

5.2.1.2 HAL (Hardware Abstraction Layer)

The HAL is a software layer that interfaces to the underlying hardware. This layer shields access to hardware functions through simplified or standard interfaces that isolate the hardware complexity from the software developer. The HAL consists of three sub-layers: the access shielding, register shielding and functional shielding layers. The access shielding layer contains the mapping between the variable names and the addresses of the memory locations; macros enable the use of the HAL in simulations. The register shielding layer enables access to the memory locations using read and write functions. The functional shielding layer allows higher layers to use a device, without knowing all the details of the device implementation.

5.2.2 HdS Terms

Term	Definition
Boot	Short for Bootstrap.
Bootstrap	The minimal set of software routines necessary to initialize the hardware to a known stable state.
Code Re-Entrancy	The ability for code to be interrupted and reinitiated with new data without destroying the interrupted execution, such that upon completing the interrupting task, the previous task can be continued to completion. Unlike task switching, re-entrant code may not imply complete storing of state.
Constant Bit Rate	An I/O port that transfers data at a constant rate. For example, with analog-to-digital converters and digital-to-analog converters, they convert data at a specific constant rate.
Context Switch	The computing process of storing and restoring the state of a CPU (the context), such that multiple processes can share a single CPU resource. Also called process switching or task switching.
Data-Driven Scheduling	The ability to schedule a task in response to the availability of data. See Scheduling Models.
Deadlock	Two or more tasks that require information or triggers from each other before proceeding. Deadlocks usually halt the operation of systems until timeouts restore operation.
Debug Port	A port used to debug the board without requiring any application running on the hardware. Normally implemented as a JTAG port.
Deterministic	Real-time implementation is deterministic if it minimizes the time to react to an incoming event such as an asynchronous interrupt, and to schedule an appropriate task that is associated with the incoming event. A deterministic system responds within a guaranteed time, not necessarily the fastest time. For example, an RTOS task-switch or an interrupt response is guaranteed as the worst possible case.

Term	Definition
Device Driver	A software module that shields the details of a particular hardware element (device) and provides a programming interface. For example, a task that handles the protocol of an I/O device or DMA channel to perform the requested transaction.
DMA Channel	A device with a device address that handles the detailed bus protocol for a transaction between an I/O device and memory, without the intermediate intervention of the processor.
Embedded System	A system designed for a specific purpose, generally fixed at the time of manufacture. Typical embedded systems include: cell phones, internet routers, satellites, GPS units, and automotive engine controllers.
Embedded Software	Software specifically designed to run on an embedded system. This includes both application and system-level software.
Event	1. A method used for task synchronization. 2. An action that causes a state change in the system being modeled. When used in the context of event driven models, events are triggers or transactions that cause the execution of the appropriate tasks to handle them.
Firmware	Embedded software that is stored as machine code within a ROM.
Fixed Instruction-Set Processors	Processors without user definable instructions. For example, all instructions are known and are described in the user manuals for the processor.
FLASH	Electrically erasable, programmable ROM.
HAL	Hardware Abstraction Layer. A functional HdS layer that shields the actual access to memory locations and allows software clients to use a device, or hardware platform, without requiring in-depth knowledge of the device.
Input/Output Control	Device driver function.

Term	Definition
Instruction-Set	A specification detailing the commands that a computer's CPU should be able to understand and execute, or the set of all commands implemented by a particular CPU design. Also called an instruction-set architecture (ISA).
Interface Timing Characteristics	The relationship and temporal order of the signals required for the proper operation of an interface.
Interrupt	See interrupt request, interrupt service routine.
Interrupt Request	A method for a peripheral device to signal to the central processing unit (CPU) that it is in need of attention.
Interrupt Service Latency	The time it takes to respond to an interrupt. That is the elapsed time from the interrupt request to when the system is available for the next interrupt request.
Interrupt Service Routine	A software routine that responds to interrupts and schedules the appropriate device driver.
Intertask Communication	The transfer of data and control between two or more tasks.
Kernel Space	The CPU has different operating levels to disallow operation at the lower levels. The operating system utilizes the CPU features to allow independent operation of programs and protection against unauthorized access to resources. Under Linux, the kernel executes in the highest level (kernel space), where everything is allowed, whereas applications execute in the lowest level (user space), where the process regulates direct access to hardware and unauthorized access to memory.
Load	Part of the HdS Control Layer and Function Control Layer. Any software routines which are responsible for copying software code to a target processor in order to be ready for execution. Load is possible from local memory devices, or (using data links) from remote data storages.
Lockout	Method of task synchronization. The prevention of access to a resource while another task has non-shared use of the resource.

Term	Definition
Memory Management	The method of (or hardware dedicated to) managing computer memory and optimizing its use. This might include techniques such as arranging used and free memory to speed access or to maximize available storage space through dynamic address translation. Additionally, the memory manager protects memory from illegal access and produces an exception error if an illegal access is attempted.
Message Queues	Queues that are used to store and retrieve multiple packets of data. This is a mechanism used to handle inter-task data communication and synchronization. Either fixed or variable length size of queues can be used to hold messages among tasks. Typically, message queues eliminate the temporal interlock between the task sending the messages and the task receiving the messages, until the queue overflows.
Microcoded Instruction-Set	Processor instructions that internally are implemented by groups of lower level instructions. These instructions are typically broad, multi-field instructions where all fields are executed in parallel.
Multiprocessor Platform	An embedded platform that contains more than one Microprocessor or DSP.
Multitasking	A technique used in an operating system for sharing a single processor between several independent jobs.
Multithreading	Sharing a single CPU between multiple tasks (or threads) in a way designed to minimize the time required to switch threads. Multithreading differs from multitasking in that threads share more of their environment with each other than do tasks. Threads may be distinguished only by the value of their program counters and stack pointers, while sharing a single address space and set of global variables.
Mutex	A mutual exclusion (mutex) object that is created so that multiple program threads can take turns sharing the same resource, such as access to a file.

Term	Definition
Netlist	A description of hardware that consists of blocks and the signals between them. It is a list of all the components and their interconnections that is typically described in a standard format, such as EDIF (Electronic Design Interchange Format), or the structural constructs of an HDL (Hardware Description Language).
Off-Line Test	1. A set of HdS functions to allow test segments to run, which are dedicated to testing ASIC, SoC, board, or system level. 2. Test segments that perform the tests.
PLA-coded Engines	Processors whose control is performed by cycling through a state diagram encoded into a PLA. Typically, the PLA controls a state register and the data path, and in turn is controlled by the state register, data, and control information from the data path.
Page Fault	A fault that occurs in a virtual memory system when a portion or page of virtual memory is not resident in real memory at the time it is required to be accessed by the processor. This then triggers a page fetch operation.
Pipes	The buffers that can be written to by one process and read by another.
Polling	The continuous checking of other programs or devices by one program or device to see what state they are in, usually to see whether they are still connected or want to communicate.
Portability	1. The ability to transfer the software from one hardware platform to another while preserving its functionality. 2. A property of software that can be ported and made to run on a new platform and compiled with a new compiler.
Pre-Emptive Scheduling	Scheduling model. The scheduler can interrupt and suspend or (swap out) the currently running task in order to start or continue running (swap in) another task.
Priority	The level of importance of an event or task.

Term	Definition
Protocol	The rules of temporal ordering of signals to synchronize the transfer of information between asynchronous events or tasks. An interrupt request is part of the protocol of the transfer of data between an I/O device and a processor. Semaphores are a specific protocol for the transfer of information between two independently scheduled tasks.
Rate Monotonic Analysis (RMA)	A collection of quantitative methods and algorithms that allow engineers to specify, understand, analyze, and predict the timing behavior and throughput requirements of real-time software systems, thus improving their dependability and ability to evolve. RMA is based on assumptions regarding software task expected execution times, and computes limits on processor loading.
Real Time	A system is said to be real time if it has critical timing requirements that must be met in order for the application to be successful.
Queue	A first-in, first-out memory, used as a transport medium.
Scheduling Models	Different algorithms used to schedule events within a system, including the following: • Priority based: each task is assigned a priority and the task with the highest priority is scheduled. • FIFO based: tasks are executed in the order in which they are ready to run. • Round Robin: tasks are executed for a specific quantum of time or time slice, at which point they are placed at the back of the run queue and the next ready task in the round-robin queue is selected. • Fair Share: an algorithm that takes into account how long a task has been blocked. The longer a task has been blocked, the more weight is given to this task, and, when it is readied, it is more likely to run. It is common to have combinations of these scheduling algorithms in order to have a system that is both real-time and responsive.

Term	Definition
Semaphores	The method for restricting access to shared resources (such as storage) in a multiprocessing environment. A specific protocol for transferring data between two independently scheduled tasks.
Shared Memory	Memory that is accessible by two or more processors.
Shielding Layer	A software layer that hides the hardware access and functionality through simplified or standard interfaces that isolate the hardware complexity from the software developer. The HAL consists of three sub-layers: the access shielding, register shielding and functional shielding layers. The access shielding layer contains the mapping between the variable names and the addresses of the memory locations; macros enable the use of the HAL in simulations. The register shielding layer enables access to the memory locations using read and write functions. The functional shielding layer allows higher layers to use a device without knowing all the details of the device implementation.
Sockets	A Unix mechanism for creating a virtual connection between processes, both locally and on other networked systems.
Software Layering	The layered software architecture recommended by VSIA, and others, to create a method for software reuse. This is one of the HdS taxonomy axes.
Stack	A data structure for storing items which are accessed in last-in, first-out order. The most common use of stacks is to store subroutine arguments and return addresses. The stack resides in memory and is accessed by a stack-pointer.
Starvation	Occurs when a task does not meet its real-time requirements because the task is either not getting enough processor time to complete its intended operation, due to other tasks taking priority, or the task does not have the data necessary to complete its intended operation.
Synchronization	Temporally aligning the transfer of information between two or more tasks or blocks of hardware. The concurrence of events with respect to time.

Term	Definition
Task	An independent thread of execution that may synchronize or communicate with other tasks. An independently scheduled software module that performs a specific function.
Task-Switching Latency	The time it takes to transfer control from one task to another independently scheduled task. The elapsed time from the halt of execution of one task to the continued execution of another task. This typically involves saving the state of the first task and re-establishing the state of the next task before transferring control.
Thread	A sequence of instructions or data pertaining to a specific task. An instruction thread is a sequence of instructions whose external interlocks have been resolved. A data thread is a sequence of data that is transferred from a task or location to another task or location with its order preserved regardless of the method of transfer. Threads also refer to multiple instruction streams which can be mapped to a single or group of processors by a multiple-threading operation system; these threads often share a single global memory space.
Timeout	An interrupt whose priority is higher than the current event in a process that was issued at some defined interval of time after the start of the event. A timeout typically initiates some action to cancel the last event, thereby eliminating any potential deadlock.
Timer	A clock that interrupts the process at known periodic intervals. A set of functions that announce each clock tick to the OS, set and obtain the current date and time, and send events to calling tasks at or after the time interval.
Trigger	A specific signal that initiates an event and triggers an action. In hardware, it is typically a single control signal that initiates a hardware operation such as capturing or sending data. It is automatically fired when a specific operation occurs.

Term	Definition
User Space	The CPU may have different operating levels to disallow certain operations at certain levels. The operating system utilizes CPU features to allow independent operation of programs and protection against unauthorized access to resources. Under Linux, the kernel executes in the highest level (kernel space), where everything is allowed, whereas applications execute in the lowest level (user space), where the process regulates direct access to hardware and unauthorized access to memory.
Variable Bit Rate	1. Information that is represented in a digital form by clusters of bits (rather than a constant stream of bits) is said to have a variable bit-rate. Most data applications generate VBR traffic. 2. An I/O port that receives or sends data at different rates. Most networks transfer packets of information at variable rates based on demand.
Watchdog Timer	A device that performs a specific operation after a certain period of time if something goes wrong with an electronic system and the system does not recover on its own.

Table 5-1 HdS Terminology

5.3 HdS Taxonomy Axes

5.3.1 Introduction

Taxonomy axes are a mechanism to allow classification, analysis and differentiation between different models, objects or types of things. This chapter presents several, more or less orthogonal, axes that encompass different requirements and aspects for the representation and roles of hardware-dependent software. The axes of the HdS taxonomy are the following:

- Life cycle axis
- Run-time and real-time axis
- Hardware architecture axis
- Software layering axis

These axes are described in detail in the following sections.

5.3.2 Life Cycle Axis

The life cycle axis defines the role and existence of HdS in relation to the life cycle of a product. The purpose of this axis is to define discrete phases in software's life cycle, with requirements for certain input and output deliverables. The life cycle covers not only software development, but also its use and reuse until the software is no longer used.

5.3.2.1 System Development

This is the system-level modeling and validation of functionality that will become software and hardware. The software exists as models, specifications, and interfaces, but not yet as production code.

Inputs:	Requirement specification
Outputs:	Executable specifications, performance models, DSP algorithms, protocols

5.3.2.2 Software and Hardware Co-Development

This is the process of implementing and verifying specified functionality with software and hardware. The software comes into existence in source-code format. Software code is either functional code to be incorporated into the final product, or software to test and bring up the underlying hardware platform.

Inputs:	Requirement specifications, executable models, hardware descriptions
Outputs:	Implemented software, documentation
Tools:	OS simulator, ISS models of processors, netlist, RTL or behavioral models of hardware, hardware emulation, fast prototypes, code generators

5.3.2.3 Debug and Optimization

This consists of debug or optimization of software on selected hardware platforms, either virtual or real. Debug of hardware with either functional, or test software. The software exists in source code and executable formats. The software passes through one or several transformations, either manually by the designer or by tools (code generators and code optimizers). The result is machine code that is ready to be delivered with the product.

Inputs:	Software code
Outputs:	Clean software code
Tools:	Debuggers, code optimizers, code generators

5.3.2.4 Use

This consists of running the actual product code. The software exists in machine-executable format.

Inputs: Machine code
Outputs: None

5.3.2.5 Retargeting

This is the process of redesigning software to run on a different processor or hardware platform. The phase may need to visit all of the previous phases of the life cycle.

Inputs: Original software
Outputs: Original software running on a new platform, thus incorporating required platform-dependent changes

5.3.2.6 Variant or Derivative Development

This consists of developing a new product by removing features from or adding features to an existing product, or by improving performance. It includes redesigning hardware to execute a different software suite.

The phase may need to visit all the previous phases of the life cycle.

5.3.2.7 Reuse

This entails reusing parts of existing products to create a new product.

Inputs: Parts of existing products
Outputs: Parts adhering to reuse guidelines and standards

5.3.3 Run-Time and Real-Time Axis

The run-time and real-time axis presents distinct modes of HdS in embedded systems, in the function of state of execution. The run-time phases require the existence of various software modules or certain platform support. Real-time phases add temporal restraints and requirements to the execution.

5.3.3.1 Run Time

The run-time axis presents the software operating environment and requirements from the perspective of sequencing and state machines.

- Boot
    - Software load

  - Hardware load (FPGA code, nano-code, DRL)
  - Software modules: boot code, boot loader
  - Platform support: flash controller, hard disk drive controller, and so on
- Configure
  - Software modules: Initialization code
- Execute
- Reload
  - Morphing code
  - Reconfigurable hardware
  - Caching Load segments of code from FLASH
  - On-target monitor or debug channel

5.3.3.2 Real-Time

The real-time axis presents the software operating environment and requirements from an execution real-time perspective (not a state perspective). Even the software layers, which do not directly communicate with hardware, may depend on hardware for scheduling or timing.

- Scheduling models
- Pre-emptive scheduling
- Data driven scheduling
- Real-time, non-real-time
- Interface timing characteristics
- Constant bit-rate port (ADC, DAC)
- Variable bit-rate port (MPEG)
- Packet port (Ethernet)
- Real-time debug access
- Sequence of operations
- Trigger conditions and states

5.3.3.3 Communication Mechanism

The intent is to list abstract communication mechanisms, such as register-mapped, message-based mechanisms, and to describe how they appear in HdS. Operating systems have a number of possible inter-task communication mechanisms. If tasks reside on different hosts, the abstract communication must pass through a physical communication medium, with mapping from abstract to physical to abstract in software adaptation layers. In fact, this situation may also occur with software-hardware communication, when the software task is provided with an abstract communication interface to hardware.

A good example for this kind of mapping is the link handler. The link handler can make tasks running on different hosts seem like they are running on the same host (from a communication point of view). The link handler of the sending host takes a message (signal) sent from task A, and then maps it, for example, to copy data to a shared RAM and to an interrupt. The link handler of the receiving host is invoked at the interrupt to copy the data from the shared RAM, and to recreate the message and send it to task B.

Software can also send a message to a hardware block. In this case, the process is similar, but without re-creating the message on the receiving end.

The thickness of the adaptation layer (link handler, device driver, and so on) depends on the abstraction and complexity of the software communication mechanism.

Communication mechanisms are listed here in a rough order of complexity:

- Address mapped: Maps directly to hardware bus transfers
- Packet based: Maps to routing and buffer copying
- Message based: Maps to buffer copies (address mapped) and hardware synchronization

5.3.4 Hardware Architecture Axis

Hardware architecture presents different aspects of the hardware platform, seen from the software point of view. (This is a link to platform-based design.) The hardware-dependent software hides or abstracts the hardware.

5.3.4.1 Architecture Synopsis

From a software perspective, hardware appears to perform a specific function. However, from a hardware perspective, the function may consist of a variety of programmable elements. The degree of programmability appears to form a continuum from pure hardware to full processor-related instructions.

For simplicity, we have selected a set of points from programmable logic (FPGAs) to traditional processors, as shown in Figure 5.2. The figure indicates that SoCs could contain arrays of small compute engines of differing types. As one moves from software toward hardware, improvements in performance can be obtained by building specific functions into firmware, by programming at a lower, finer-grained levels of hardware operations, thus eliminating the unnecessary movement of data that occurs in standard processor code. Alternatively, one can add instruction extensions to tailor a general-purpose ISA to specific applications. These instruction

extensions may be implemented as special purpose hardware, but still made available to software developers via standard C compilers. Conversely, programmable solutions often run at slower clock frequencies than custom-designed processors because the lower-level operations are more programmable.

While the following four levels of processors can exist separately, it is also common to have multiple levels within one processor, as shown in Figure 5.2. Such processors could be considered composite processors. Still, when mapping the composite processor onto the existing levels, it is typical to refer to the processor at the level to which users have the most access. For example, a processor with microcode that is not accessible to the programmer is considered an instruction code-level processor, whereas a processor with microcode that is directly accessible and modifiable by the programmer, particularly if all but the most instructions are available in microcode, is considered a microcoded engine. Similar examples can also be drawn for the other levels.

To elaborate on this idea further, we articulate that the instruction-set processor (for example, GPP) is based on published instruction sets. Below this layer, another classification is called out and named microcode processors, on which instruction sets can be built. Some processors can use both the microcode and the instruction sets. While these can be called out at the lowest level that they employ, for example, the microcode, the suggestion is to classify them under a new term called composite, which includes more than one of the classifications. A processor could certainly be built from these composites. The extensible processor based on a standard instruction set sits in between these four categories.

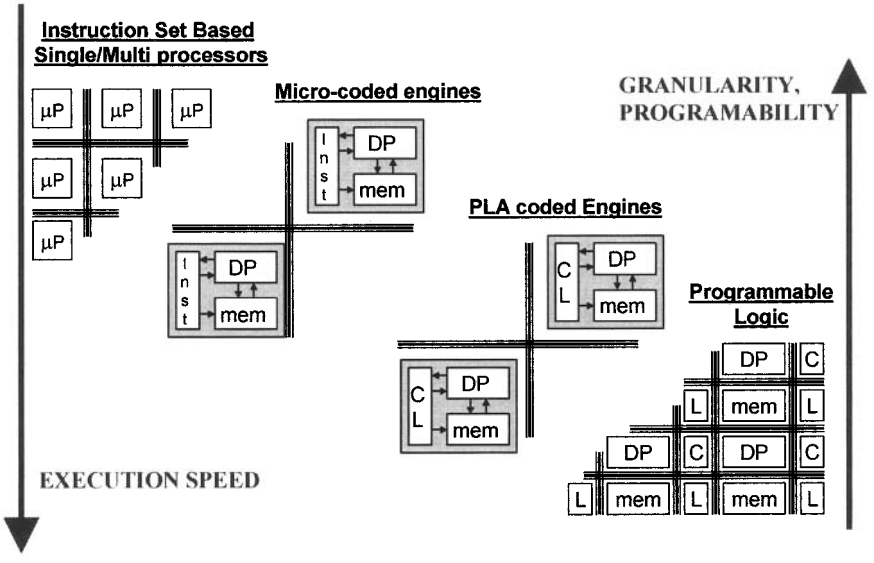


Figure 5.2 Architecture Synopsis

Programmable Logic Code

Programmable logic code consists of two large blocks of memory; one for bit-selection of the programmable functions, and one for bit-selection of the wire segments between the functions. In traditional FPGAs, the functions are defined by the contents of small single-bit wide memories, and the interconnects are traditionally defined by a sparse matrix where only the actual wire segment connections are set. It is sparse because most configurations use less than 5% of the available connections in the FPGA. In Figure 5.2, some of the elements could be hardwired datapath elements that require much less memory to define the encoded functions. In addition, the data could be routed on a bus-wide basis, which would also require less memory for the encoded wire-segment connections.

The program is usually completely loaded into the two blocks of memory before the execution of the function can begin. Typically this is done serially at power up. In these cases, the resulting programmable logic code is indistinguishable from hardware.

The high-level language for this code is typically RTL, which is synthesized, placed, and routed, and then converted into the loadable blocks of memory described previously.

- Control characteristics:
  - Use wired control and data-path elements
  - Hide functional logic
- Data characteristics: Signal level

- OS requirements: None

PLA Code

A PLA-coded engine typically includes some memory, datapath elements, and a PLA. A PLA is a matrix of AND and OR functions driven by external data and control lines, which, in turn, drives feedback registers, the controls for the data-path elements, memory, and outputs.

A PLA has a bit matrix memory for its AND and OR planes, and is typically loaded in the same manner as the programmable logic code. The matrix is sparse because all inputs connect to all outputs on each plane, which is usually not the case with the actual logic. Still, it is usually less sparse than programmable logic code because the routing information is both limited and combined with the functional description.

High-level languages typically describe state machines defined by sets of Boolean equations. These are converted into the AND and OR functions within the PLA, given the physical locations of the signals. The registers contain the current state and other Boolean data.

- Control characteristics:
    - Usually single cycle operation
    - Usually state machine logic
    - Architecture specific
- Data characteristics:
    - Protocol aware
    - Single-word operation
    - Architecture specific
- OS requirements: Simple loop waiting

Microcode

Microcoded processors typically have wide microcode instructions, where all the decoded fields (with the exception of memory addresses) for the entire datapath subsystem are available in the microcode instruction. Unlike PLA or programmable logic, the code is essentially sequential, where the processor executes one instruction per cycle. Unlike RISC code, many parallel operations may be executed by one microcode instruction.

Typically, microcode has minimal stalls or locks on its execution by the processor, and is never executed out of order.

- Control characteristics:
    - Usually decoded instructions
    - Usually single-cycle operation
    - Usually architecture specific

- Data characteristics:
    - Usually word-level transfer
    - Usually protocol hidden
    - Single-word operations
    - Architecture specific
- OS requirements: Simple scheduling and loading

Instruction-Set Code

RISC or CISC is the instruction code for most processors today. Traditionally, RISC code was designed to execute at a rate of one instruction per cycle. However, current processors (such as VLIW, superscalar) may execute multiple instructions at a time, and may even execute them out of order, as long as the resulting register contents are correct at the end of the software operations. The instructions, while simple, are highly encoded, requiring only as many bits as necessary to define the memory locations and operations to be performed.

- Control characteristics:
    - Single or multiple instructions
    - Usually encoded instructions
    - Usually multi-cycle operation
    - Hides Architecture and processor function
- Data characteristics:
    - Packet burst
    - Protocol hidden
    - Multi-word operation
    - Hides architecture
- OS requirements:
    - Loading, scheduling, and interrupt handling
    - Varies depending on memory

It is important to note here that software engineers may not see the previously described classification as clearly as hardware engineers. In the software world, it may be more natural to describe the classification as processor instructions (which break down into either general-purpose instructions or application-specific instructions), and at a layer below as single-or multi-cycle operations. In this context, it is important to point out that in the software world, there are architectures known to augment instructions (application specific or generic) with hardware functions having the ability to load and execute functions. In some cases, this is called microcode; in other cases it is not.

5.3.4.2 OS Requirements

The level of OS requirements is determined by the complexity of the system and the problem being addressed. The following sections give a simple taxonomy of the processor capabilities and the corresponding OS requirements. While some increasing level of hardware capability in an MMU is required for each of these memory levels, the amount of OS support varies, depending on the capabilities of the MMU. For example, implementing virtual memory and protected memory is best done by using an MMU. It is also possible to build systems without interrupts. In such cases, a simple polling process checks the status of all I/O locations, and then processes the status before going back to the polling.

Memory

Simple Memory Mapping

In this case, the processor accesses devices using specific memory addresses.

Protected Memory

With protected memory, an internal interrupt puts the processor into protected memory. The OS must now include a separate initiator of tasks, and the tasks must request initiation of I/O operations and new tasks through the operating system.

Secondary Memory

If the tasks reside outside main memory, a loader must also be added to the OS to be able to initiate non-memory-resident tasks.

Virtual Memory

A virtual memory system requires an additional function in the OS to manage the virtual memory space. Typically, a virtual-memory system includes a dynamic-address translation (DAT) mechanism to provide the actual physical address when given the virtual address. This is usually done in hardware. The management of the DAT and the available virtual memory space is usually performed in a virtual-memory management subsystem of the OS.

Bank-switched Memory

Some processors have limited address space that is smaller than required by the application. In such cases, the memory is arranged in banks that are equal to or smaller than the inherent address range. The system then uses some sort of redirection mechanism to switch to the proper bank as needed.

Interrupts

None

A simple polling scheme is usually used in systems without interrupts.

External Interrupts

In this case, the OS should have an interrupt-handling subsection and drivers to handle the interrupts. A simple loop can be used in place of a scheduler, but tasks must be initiated in response to an interrupt. Conversely, a more complex scheduler can be used, but this is limited to operating during interrupts.

Internal Interrupts

In this case, the processor can interrupt itself when error conditions, traps, or exceptions occur within the task being executed. Additionally, in a multi-tasking environment, the OS could have a scheduler to correctly schedule the next task.

Real-time Clock

With a real time clock, the scheduler may not only initiate tasks, but can also stall them when a higher priority task must be executed. The OS updates the task list, and schedules the highest priority task on every time tick. With a real-time clock and the appropriate RTOS, independent tasks can be scheduled to execute at periodic or specific times as needed by the system.

5.3.4.3 Architecture of Software Defined by Hardware

CPU Subsystem

The CPU subsystem contains the processor core, bus interface, floating-point unit, debug support, and peripherals. The processor core contains registers, memory addressing, cache, exceptions, and interrupts. The software sets up the CPU subsystem at boot time.

CPU Registers

Different processors have their own sets of registers. The software sets up these registers as indicated by each processor's specification.

Address Space

The address space can be either a physical address space or a virtual address space. If the processor supports a Memory Management Unit (MMU), the virtual address can be translated into a physical address using techniques like the translation look-aside buffer (TLB). Alternatively, conversions from physical to virtual, from virtual to physical, and from one type of bit addressing to another (such as 32-bit to 64-bit, or 8-bit to 16-bit) are typically handled in software.

Cache Organization

Each processor may have several on-chip caches for instruction and data. Cache sizes are configurable. The software sets up cache memory, and can disable or enable the cache when needed. Caches may be organized into levels of different capacities and access times (L1, L2 and L3 caches are now common) in order to mask long off-chip memory access latencies. In addition, multiple data caches may make sense for DSP style processors or processors with DSP oriented functions which want efficient X-Y memories.

CPU Exceptions

Each processor has exception handling. The typical exceptions include the following: address error, bus error, interrupt, and floating-point exception. The software sets up exception vectors and handles each exception properly. An Interrupt is an exception. There are two broad classes of exceptions. Internal exceptions occur as a direct result of the instruction stream, and include things like protection violation, undefined instructions, and so on. External (or asynchronous) exceptions are a result of events external to the processor, and include bus errors, timer interrupts, peripheral interrupts, and so on.

Interrupts

Each processor supports up to three types of interrupts: non-maskable interrupt (NMI), external interrupt, and internal interrupt. Based on the interrupt source, software control dispatches to the appropriate interrupt service routine (ISR).

Floating-point Unit

Each processor may support floating-point operations. The software controls the programming model of the floating-point unit.

Memory Subsystem

Memory is one of the main components in a system. There are two types of memory: ROM and RAM.

ROM includes Programmable ROM (PROM), Erasable programmable ROM (EPROM), and Electrically Erasable Programmable ROM (EEPROM). These read-only memories are normally used for BIOS or boot loaders. For many devices, ROM is also the repository of the operating software and applications such as cell phones, Personal Digital Assistants (PDAs), and so on.

RAM includes Static RAM (SRAM), Dynamic RAM (DRAM), double data rate (DDR), Ferroelectric, and Magnetic RAM. These random access memories are called memory, and are a temporary storage for program data. From another point of view, it might be more useful to characterize the RAM as volatile and non-volatile. In volatile memory, the data disappears when system power is removed, and the data remains for non-volatile memory.

Each system has different size and types of memory, and the software configures the memory controller and checks if the memory is accessible at boot time. If the processor supports MMU, the virtual memory is supported using paging and segmentation. A good example for virtual memory is the Linux operating system. The virtual memory allows the system to separate program (virtual) addresses from actual (physical) addresses.

I/O Subsystem

Each SoC may support peripherals for input and output. Based on the application domains, the devices for an I/O subsystem could include examples such as Tuner, IR, IEEE1394, USB, serial port, parallel port, video and audio coder/decoder, and many others. For the application, OS, or other drivers to access the IO devices, the device-specific software (device drivers) must be written to provide the method for accessing the device.

5.3.4.4 Multiprocessor Architectures

Multiprocessor systems require additional complexity to synchronize the activities on several processors in the system, as well as to manage the communications between these processors. Implementations may consist of homogeneous processors for additional processing power of similar,

possibly identical processes. They can also include heterogeneous processors in which different processors are used for their specific characteristics such as DSPs for signal processing and a RISC for network communications. Additionally, there are interactions with the memory subsystems that affect the method of the overall multiprocessor implementation. For example, memory can be used for communications between processors and for scheduling or synchronizing tasks across processors.

The intercommunication between processors consists of a physical layer that allows processors to send messages between their domains. This may consist of a FIFO, dual-port RAM, or some other communication mechanism.

The following sections contain several examples of multiprocessor architectures.

Example 1: General Description of System-message Exchange

The system architecture consists of one master processor and many slave processors. The master processor has access to all of the memory contained in each slave processor domain. Each slave processor can interrupt the master and the master can interrupt the slave. At system startup, the master processor loads each slave with its assigned program. Once the program is loaded, the slave processor awakes from reset and requests from the master its configuration record. It does this by using a message-exchange protocol.

The following is an example of a normal exchange:

WHOAMI	-Slave #1 requests configuration from master.
RTS	-Slave #1 requests to send message to slave #2.
CTS	-Slave #2 acknowledges request is available for receipt from slave #1.
SENDM	-Master moves the data between slave #1 and slave #2.
SENDEOM	-Slave #2 acknowledges successful receipt to slave #1.

The message-exchange protocol allows for each slave processor to interrupt the master when a message is ready to send. The master reads the header of the message to determine its destination, and then makes sure the destination is ready to receive the message. The master then sends the message by DMA to the appropriate processor, and then interrupts the destination slave.

Example 2: Hardware FIFOs

This system does not interconnect its processors in a typical bus fashion. Rather, its backplane is made of a system of FIFOs controlled by a state machine. A slave processor ready to send data to another processor requests

a FIFO from the state machine. The state machine scans the backplane looking for messages that are queued and ready to transmit. Once a source slave is selected, its destination slave is queried to ascertain if the FIFO can be connected to it. If the destination slave is busy connected to a different transmitting slave, the transaction is canceled, and the state machine continues to scan the bus looking for messages to send. Eventually, the state machine connects a willing transmitter with a willing receiver, and the message is transferred without any processor intervention. Finally, each processor is interrupted to show that the message was transmitted and received. This allows for multiple messages to be queued, sent, and received simultaneously, with minimal delay in the hardware and running at the speed of the state machine, not a master-CPU running code.

Example 3: Typical Shared-memory System

A loosely coupled system utilizes shared memory to pass messages between processors sharing a common bus. This interface is treated as if it were an Ethernet controller, and routes messages appropriately. Shared-memory systems, where multiple CPUs have at least partial access to a common memory pool and an ability to interrupt the CPU that is acting as an arbitrator, can utilize TCP/IP to pass messages between CPUs. For example, a VME card cage with six Motorola 162 VME cards can be recognized in the system from one Ethernet port, and one can pass appropriate messages to each of the VME cards. The VME card with the external Ethernet interface also acts as a router for the entire system. Messages sent between card #2 and card #4 are passed through card #1 as with an Ethernet router.

5.3.5 Software Layering Axis

The software layering axis presents a layered model for embedded software, as well as the HdS API as one of the key concepts in the HdS domain. First, this section describes the basic model. Then, it gives additional information about layering. The intent is to describe how software layering aids in software reuse by abstracting the hardware to create an easy-to-understand and standardized programmers model.

5.3.5.1 Basic model

The Basic model defines a layered architecture for the API. The hardware abstractions and functions defined in each layer are responsible for the following:

- Communicating with the peer HAL-API layers for each module, core, or circuit running in the SoC
- Providing services to the layer above

The objectives of layered APIs include the following:

- To decompose the API into understandable objects or layers
- To provide standard interfaces between hardware, drivers, APIs, and operating systems
- To provide symmetry in functions performed at each module, core, or IP object in the SoC
- To provide a means to predict behavior and control the effect of any changes made to the API or SoC
- To provide standard terminology to facilitate communications in the community of designers, developers, managers, vendors, and users

The Hardware Application Layer

The HAL is composed of a framework that is depicted in Figure 5.4. It is composed of the following major building blocks: access shielding, register shielding, and functional shielding.

- **Access shielding** shields the actual access to the hardware. This is the layer of the software where one maps software and HdS to the hardware platform, which may also be a (virtual) simulator for the hardware platform, such as a VHDL simulator, a co-simulation or co-emulation (VHDL on emulator, software on a Workstation) environment, or an emulator giving a virtual hardware platform or a rapid prototype. The key goal for this layer is to ensure that the software lying above the access shielding layer (Register shielding, functional shielding, application software, and so on) has no impact on the (virtual) hardware platform. The user should not have to know how the actual physical access is being performed. For example, if one wants to read a register that is mapped in the IO space, the read access done by the user is transformed by this layer to an actual IO access without requiring the user to understand the peculiarities of every type of access.
- **Register shielding** makes the software layers above it (functional shielding, application software, and so on) independent of the physical addresses of the registers. It also enables the superior software layers to use only logical names for these items. This decouples this layer from the superior software layers, and makes them independent of physical address changes. For this purpose, a

database is used that translates the logical given names to the actual physical addresses.

- **Functional shielding** is an HdS layer that groups certain functionalities in a small, limited API that is offered to the superior software layers to respond to their needs. The goal of this layer is to give a simpler, more abstract API to software designers of all software lying above the functional shielding layer. Such an API is both more abstract and easier to understand. This layer is composed of some control code (for example, a finite state machine (FSM)), registers, and other kinds of accesses. A basic example of an HdS API mechanism that is part of this functional shielding layer is the ***init*** mechanism. Such a mechanism initializes a virtual component on an SoC into a reference mode without exposing to the user the complex register accesses that need to be done in a certain order in order to accomplish this.

5.3.5.2 Layers Included in the Layered API Model

Hardware Layer

The lowest layer in the HdS model is called the hardware layer. The functions within this layer are responsible for starting, stopping, single stepping, running, testing, and observing a physical circuit, core, or module of the SoC. This level contains the access registers and the physical implementation of interrupts.

Primitive Function Layer

The primitive function layer provides the lowest level of shielding, access shielding. Register shielding is also used here. This layer is responsible for the transfer of control or data information from the hardware layer to the inter-API communications layer or the interface layer. This layer provides information about the hardware for the next layer interface.

Inter-API Communications Layer

The inter-API communications layer specifies the interface, the routing, and the communications between the cores. It provides control to ensure that the cores, modules, or circuits do not become overburdened. One of its most important functions is to detect hardware errors and provide a recovery mechanism.

Interface Access Layer

The interface access layer provides the register and functional shielding. This layer has software-callable routines that aggregate the primitive functions into low-level operations from which drivers, control functions, initialization, and debug-access software functions can be created. This layer is designed to keep the user isolated from some of the physical and functional aspects of the hardware.

OS Layer

The OS layer is the Operating System.

Application Layer

The application layer is the user application.

5.3.5.3 Control, Data, Hardware, and Software Layering

Figure 5.3 helps bridge the software and hardware worlds with common terminology, and shows how they fit within each other's disciplines:

- Objects and classes: Using existing data formats, hiding computational complexity
- Code layer:
    - Application threads
    - Using existing API, hiding scheduling complexity
- RTOS layer:
    - Scheduler and services
    - Using existing transaction interface, hiding all timing complexity

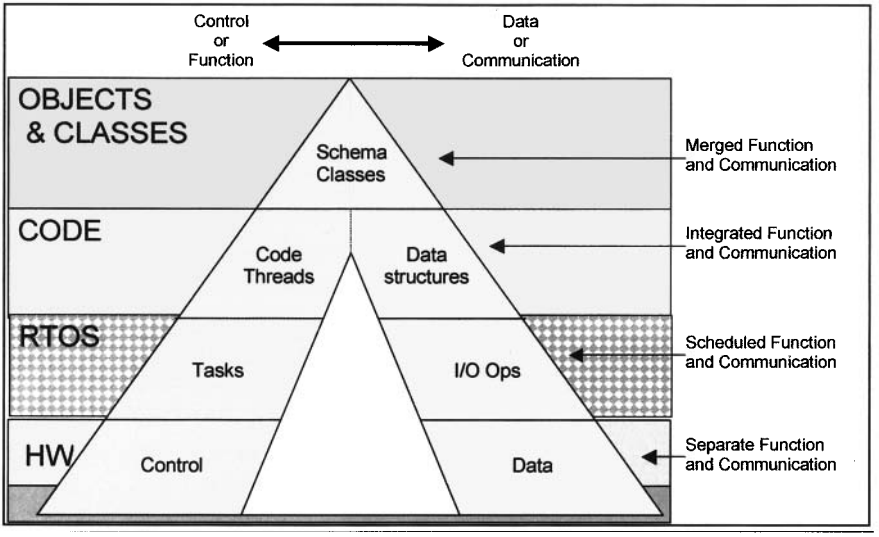


Figure 5.3 Control and Data Merge in Software Domains

For example, at one extreme (shown at the base of the triangle in Figure 5.3), it is common to call out the data–for example, the components being acted upon and the control, the instructions to act on the data. However, as one moves more into the software world (shown at the top of the triangle in Figure 5.3), the separation between data and control is less distinct, and the emphasis is on abstracting the data and the control associated with a single object, or class, inside the object or the class.

While these two disciplines have different goals, it is important to understand each other's needs because these models are trying to address exactly this interface.

Between these extremes, there are two layers, the RTOS (Real-time Operating System) and application code, as examples of layers that may help bridge the gap.

In Figure 5.3, the communication lies between the data and the control (identified by the white triangle in the middle). In the hardware world, the communication between data and control is accomplished with signals, while at the higher levels, these are accomplished with protocols.

5.3.5.4 HdS API

Figure 5-4 shows the relationship of the HdS as a bridge between device drivers and the SoC. This section describes the relationships of the HdS and HdS API to the complete SoC-based system, including the hardware and software. It provides a view of the Hardware Abstraction Layer (HAL) as

part of the HdS, and its relation to the hardware layer, OS layer, and application layer.

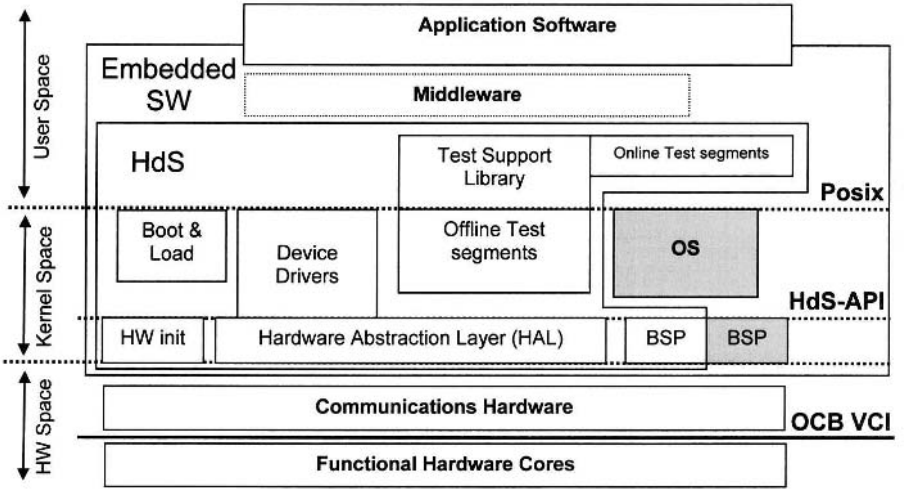


Figure 5.4 Hardware Abstraction Layer (HAL) and HdS API

The layers introduced in Section 3.5.1 can be mapped onto hardware space (the hardware layer), kernel space (the primitive function layer, the inter-API communications layer, the interface access layer, and the OS layer), and user space (the application layer). While the interface between kernel space and user space could be standardized using the POSIX interface [ISO 9945], there are other interesting standards. For example, VSIA has defined the OCB (On-Chip Bus) VCI interface for standardizing hardware-communication interfaces. Inside the kernel space, those parts that are directly dependent on hardware register maps and hardware interrupt structures can be separated from functional parts such as device driver logic, offline test segments, generic boot, and loads. This separation is done using the HdS API. All parts below the HdS API (the HAL, BSPs) carry hardware platform-specific information, and can partially be derived from the physical hardware structure in terms of register maps. The kernel space parts above the HdS API implement functional logic, and are independent from implementation details of the underlying hardware platform. Parts of the hardware-dependent software, as defined in Section 2.1.1, cover the user space as well. Examples include functional test segments, or test support libraries, that serve both hardware and system tests.

Figure 5.5 shows further relationships between these entities.

5.3.5.5 Device Drivers

Each device driver may have six standard functions for supporting OS access to the hardware:

- init()
- open()
- close()
- read()
- write()
- ioctl()

These functions are normally required by an OS or RTOS as an abstraction layer to hide the details of the specific device, while allowing the system to use a common interface for like devices. Depending on the system architecture, a device driver may control more than one IP block or more than one instance of a communication channel. This allows similar devices to use a common code base with separate data objects in order to simplify the design through code reuse.

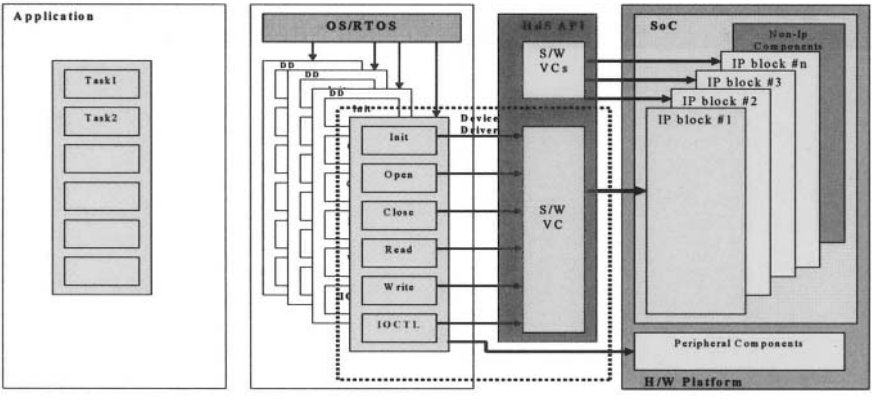


Figure 5.5 The Relation of the HdS API to Application Software, OS, and the SoC's Hardware Platform

5.4 Conclusion

This chapter has presented the beginnings of a taxonomy or classification scheme for software, software entities and hardware-dependent software in particular. Because the worlds of hardware and software design have been and remain relatively independent, even in the area of SoC where they have

collided more frequently, it is important to agree on common terminology for entities, models and classifications, and this chapter has started that process. However, further evolution in the design of mixed HW-SW systems and SoCs will no doubt advance the development and definitions of these concepts.

REFERENCES

[AND 00] Tom Anderson and Robin Bhagat, "Tackling Functional Verification for Virtual Components," *Integrated System Design*, November, 2000.

[AND 02] Tom Anderson, "Verification reuse enables design reuse," *Electronic Engineering Times*, December 23, 2002.

[AND 04] Thomas L. Anderson, "Functional Verification in the Context of Design Reuse," *DesignCon 2004 Conference Proceedings*, January, 2004.

[ARM 95] J. Armstrong, "High Level Generation of VHDL Testbenches," *Spring 1995 VIUF Proceedings*, 1995.

[ASH] P. J. Ashenden, J.P. Mermet, R. Seepold, editors, *System-on-chip Methodologies and Design Languages*, Kluwer Academic Publishers, 2001.

[BAI 00] B. Bailey, G. De Jong, P. Schaumont, C. Lennard, "Interface Based Design," *Forum on Design Languages Proceedings*, 2000.

[BAI 05] B. Bailey, editor, *The Functional Verification of Electronic Systems: an overview from various points of view*, IEC Press, 2005.

[BAY 99] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K Schmid, T. Widen and J.-M. DeBaud, "PuLSE: A Methodology to Develop Software Product Lines," *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR99)*, pp. 122-131, May, 1999.

[BLAH 85] R. Blahut, *Fast Algorithms for Digital Signal Processing*, Addison Wesley, 1985.

[CAFE] *Concepts to Action in System-Family Engineering (CAFÉ)*, ITEA Project 00004, http://www.esi.es/en/Projects/Cafe/cafe.html, 2004.

[CER 02] E. Cerny, S. Dudani, "Authoring Assertion IP using OpenVera Assertion Language," *Proceedings of the International Workshop on IP-Based System-on-Chip Design*, October, 2002.

[CHA 99] Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew McNelly and Lee Todd, *Surviving the SOC Revolution: A Guide to Platform-Based Design*, Kluwer Academic Publishers, 1999.

[CLE 02] Paul Clements and Linda Northrop, *Software Product Lines: Practices and Patterns,* Addison-Wesley, 2002.

[DAV 86] William H. Davidow, *Marketing High Technology: An Insider's View*, Free Press, 1986.

[DEB 99] Jean-Marc DeBaud and Klaus Schmid, "A Systematic Approach to Derive the Scope of Software Product Lines," *Proceedings of the 1999 International Conference on Software Engineering*, pp. 34-49, 1999.

[DICT] Denis Howe, editor, *Free On-line Dictionary of Computing*, http://www.foldoc.org/, 2004.

[DON 04] Adam Donlin, "Transaction level modeling: flows and use models," *Proceedings of CODES+ISSS 2004*, pp. 75-80, 2004.

[DONBR 04] Adam Donlin, Axel Braun, and Adam Rose, "SystemC for the Design and Modeling of Programmable Systems," *Proceedings of FPL 2004*, pp. 811-820, 2004

[ECK 92] W. Ecker, and M. Hofmeister, "The Design Cube -A Model for VHDL Design flow Representation," *Proceedings of the EURO-VHDL*, pp. 752-757, 1992.

[ESAPS] *Engineering Software Architectures, Processes, and Platforms for System Families (ESAPS)*, ITEA Project 99005, http://www.esi.es/en/Projects/esaps/esaps.html, 2004.

[FAM 94] S. Famorzadeh, et. al., "Rapid Prototyping of Digital Systems with COTS/ASIC Components," *Proceedings of RASSP Annual Conference*, August, 1994.

[FOS 03] Harry D. Foster, Adam C. Krolnik and David C. Lacey, *Assertion-Based Design*, Kluwer Academic Publishers, 2003.

[FOSBEN 01] Harry Foster and Lionel Bening, *Principles of Verifiable Verilog Design: Second Edition*, Kluwer Academic Publishers, 2001.

[FOSCOE 01] Harry Foster and Claudionor Coelho, "Assertions Targeting a Diverse Set of Verification Tools," *The 10th Annual International HDL Conference Proceedings*, March, 2001.

[GAW 02] Annette Gawer and Michael Cusumano, *Platform Leadership: How Intel, Microsoft, and Cisco Drive Industry Innovation*, Harvard Business School Press, 2002.

[HASH 99] M M Kamal Hashmi, "Virtual Component Interfaces," *Forum for Design Languages, 1999 (FDL'99)*, Lyon, France, 1999.

[IEEE 1076] *1076-2002 IEEE Standard VHDL Language Reference Manual*, IEEE, http://standards.ieee.org/faqs/order.html, 2002.

[IEEE 1149] *1149.1-2001 IEEE Standard Test Access Port and Boundary-Scan Architecture*, IEEE, http://standards.ieee.org/faqs/order.html, 2001.

[IEEE 1364] *1364-2001 IEEE Standard for Verilog Hardware Description Language*, IEEE, http://standards.ieee.org/faqs/order.html, 2001.

[IEEE 1471] *1471-2000 IEEE Recommended Practice for Architectural Description for Software-Intensive Systems*, IEEE, 2000.

[ISO 9945] *ISO/IEC 9945-x:2003, Information Technolog --Portable Operating System Interface (POSIX)*, ISO, 2003.

[JANTSCH] Axel Jantsch, *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*, Morgan-Kaufmann Publishers, 2004.

[KEU 00] Kurt Keutzer, A. R. Newton, J. M. Rabaey, and Alberto Sangiovanni-Vincentelli, "System Level Design: Orthogonalization of Concerns and Platform-Based Design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1523-1543, December, 2000.

[LEE 98] Lee, E.A. and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217-1229, December, 1998.

[MADI 95] V. Madisetti, "System-Level Synthesis and Simulation VHDL: A Taxonomy and Proposal Towards Standardization," *VIUF Spring, 1995 Proceedings*, 1995.

[MAR 03] Grant Martin and Henry Chang, editors, *Winning the SOC Revolution: Experiences in Real Design*, Kluwer Academic Publishers, 2003.

[MER 01] Jean Mermet, Peter Ashenden and Ralf Seepold, *System-on-Chip Methodologies and Design Languages*, Kluwer Academic Publishers, 2001.

[MEY 97] Marc H. Meyer and Alvin P. Lehnerd, *The Power of Product Platforms: Building Value and Cost Leadership*, Free Press, 1997

[OVL] *OVL Reference Manual*, Accellera, http://www.verificationlib.org, June, 2003.

[PSL] *Property Specification Language Reference Manual Version 1.0,* Accellera, http://www.accellera.org/pslv101.pdf, 2003.

[RASSP 98] *VHDL Modeling Terminology and Taxonomy Revision 2.4*, RASSP Taxonomy Working Group (RTWG), http://www.eda.org/rassp/index.html, February 23, 1998.

[SAN 99] A. Sangiovanni-Vincentelli and A. Ferrari, "System Design -Traditional Concepts and New Paradigms," *ICCD 99*, pp. 2-12, October, 1999.

[SAN 01] Alberto Sangiovanni-Vincentelli and Grant Martin, "Platform-Based Design and Software Design Methodology for Embedded Systems," *IEEE Design & Test of Computers*, pp. 23-33, November-December, 2001.

[SAN 02] Alberto Sangiovanni-Vincentelli, *Defining platform-based design*, EEdesign, http://www.eedesign.com/features/exclusive/OEG20020204S0062, February 5, 2002.

[SATH 01] Ramesh Sathianathan and Tom Anderson, "Assertion Methodologies for Verilog Design," *Integrated System Design*, May, 2001.

[SEIPLP] The Software Engineering Institute Product Line Practice (PLP) Initiative, http://www.sei.cmu.edu/productlines/index.html, 2004.

[SRS] *Functional Verification Semiconductor Reuse Standard V3.0*, Motorola, http://www.motorola.com, 2003.

[SV] *SystemVerilog 3.1a Language Reference Manual*, Accellera, May, 2004.

[TI 90] "*Modeling and Simulation*" Texas Instruments Semiconductor Group, 1990.

[TRUE] TrueScope Technologies, http://www.truescope-tech.com, 2004.

[VHDL A] *Department of Defense Handbook - Documentation of Digital Electronic Systems With VHDL*, http://www.combatindex.com/mil_docs/pdf/Hopper/MIL-HDBK/CI-62-MH-2399-3957.pdf, 1996.

[VSIA AMS] *Analog/Mixed-Signal VSI Extension Specification Version 1 2.2 (AMS 1 2.2)*, VSIA Analog/Mixed-Signal DWG, http://www.vsi.org, February, 2001.

[VSIA FV] *VC/SoC Functional Verification Specification Version 1 (VER 21.0)*, VSIA Functional Verification DWG, http://www.vsi.org, March, 2004.

[VSIA IV] *Soft and Hard VC Structural, Performance and Physical Modeling Specification Version 2.1 (I/V 1 2.1)*, VSIA Implementation/Verification DWG, http://www.vsi.org, January, 2001.

[VSIA OCB] *On-Chip Bus Attributes Specification Version 1 (OCB 1 2.0)*, VSIA On-Chip Bus DWG, http://www.vsi.org, September, 2001.

[VSIA SLIF] *System-Level Interface Behavioral Documentation Standard Version 1 (SLD 1 1.0)*, VSIA System-Level Design DWG, http://www.vsi.org, March, 2000.

[VSIA VCT] *Virtual Component Transfer Specification Version 2 (VCT 1 2.1)*, VSIA Virtual Component Transfer DWG, http://www.vsi.org, January, 2001.

[WAL 97] Susan Walsh Sanderson and Mustafa Uzumeri, *The Innovation Imperative: Strategies for Managing Product Models and Families*, Irwin Professional Publishing, 1997.

[WEB] Webster Dictionary, http://www.webster-dictionary.net, 2004.

[WIKI 03] *Wikipedia, the Free Encyclopaedia*, http://www.wikipedia.org/wiki, 2003.

INDEX

Abbreviations xi
Abstract Behavioral Model 39
Abstraction Level 52
Access Shielding 162
Accuracy 10, 53
Acronyms xi
Address Space 158
Algorithm Model 37
API Platform 133
Application Data 58
Application Driven 129
Application Layer 164
Application-Specific Prototype..
.....80, 82
Architectural Models 38
Architecture 56, 133
Architecture Platform 132
Architecture-Driven 128
Argos 34
Assembly Code 16, 51
Assertion 82
Assertion Monitors 82
Atomic 66
Attribute 67
Bank-Switched Memory 157
Basic Delay Model 47
Behavioral Block 66
Behavioral Model 19, 26, 82
Black-Box 15
Block-Based 128
Boot 139
Bootstrap 139
Boundary Scan 63
Branch Coverage 81, 82
Bus Functional Model 27
Cache 158
Calculus of Communicating
Systems 35
Cell 67
Cell Level Model 45
Channel 66
Checker
expected result 73, 84, 87
golden model 88, 92
protocol 73, 85
Circuit Level Model 46
Code Coverage 81, 83
types of 81
Code Re-Entrancy 139
Common Features 133
Communicating Sequential
Processes 35
Compliance Tests 83
Component 55
Component Interface 29
Computation Model Classes 32
Concurrent Processes 33
Constant Bit-Rate 139
Constraint 83
Constraint Driven 84
Constraint Driven Generation ...77
Context Switch 139
Control Data 58
Core Platform 109, 110
Coverage 81
cross 82
formal 77
Coverage Monitor 83
CPU Registers 158
CPU Subsystem 157
Cross Coverage 82
Cycle-Based Simulation 83
Cycle-Callable 43

Data Resolution....13
bit logical....14
composite....14
format....14
property....13
token....13
value....13
Data-Driven Schedule....139
Dataflow Graph....32, 41
Datum....66
Deadlock....139
Debug port....139
Derivative....132
Derivative Interface....111, 112
Design Object Classes....54
Design Rules Checks....91
Design Specification....64
Design-Tool Terms....62
Detailed Behavioral Model....43
Deterministic....139
simulation....71, 87
Simulation....83
Device Driver....140, 167
Directed Simulation....83
Discrete Event....34
DMA channel....140
Domain....133
Domain Analysis....133
Driver....83
Dynamic Formal Verification....76, 83
Dynamic Verification ...71, 83, 87
Dynamic-Formal Hybrid Verification....76
Ecker....2
Economies of Scope....133
Electrical Rules Checks....91
Embedded Software....140
Embedded System....140
Emulation....59, 62, 78, 79, 84
Equivalence Checking....92
boolean....90
formal....90
sequential....90
Equivalence Verification 70, 87
Essential Asset....133
Esterel....34
European Software Institute....106
Event....140
Event-Based Simulation....84
Exception....158
Executable Design Specification....64
Executable Specification....36, 64
Executable-Requirement Specification....64
Expected Result Checker....73, 84
Expected Results Checker....87
Expression Coverage....81, 84
External Interrupt....157
External Resolution....10
Finite State Machines....34
Firmware....57, 140
Fixed Instruction-Set Processor....140
FLASH....140
Floating-Point Unit....159
Formal Coverage....77, 84
Formal Platform Architectural Descriptions....132
Formal Verification....74, 84
FSM....34
arc coverage....81, 84
Full Functional Model....43
Functional Coverage....82, 84
Functional Model....24
Functional Resolution....14
algorithmic process....14
boolean operation....14
digital logic....14
mathematical relationship....14
Functional Shielding....163
Functional Test....63

Functional Verification Mapping 94
Functional-to- Physical Architecture Mapping....134
Gajski....3
Gate-Level Model....45
Gray-Box....15
HAL....138, 140
Hard Prototype
design options....80
parameters....80
Hardware....57
Hardware Abstraction Layer..136, 138, 165
Hardware Acceleration....72, 84
Hardware Application Layer...162
Hardware dependent Software....135, *See* HdS
Hardware Layer....163
Hardware Model....73, 85
Hardware, and Software Layering....164
Hardware/Software Co-Verification....77, 85
Hardware-dependent Software137
Harel Statecharts....34
HdS....137
API....165
hardware architecture axis..151
life cycle axis....148
real-time axis....149
run-time axis....149
software layering axis....161
HdS Taxonomy
axes....147
Hierarchy....52, 53, 131
High-Level Language....16, 50
Hybrid Model....31
I/O Subsystem....159
Information Classes....58
Input Constraint....85
Input/Output....140
Instruction-Set....141
Instruction-Set Architecture....41
Instruction-Set Code....155
Instruction-Set-Architecture....16
Integrated and Managed Features....133
Integration Platform....131
Integration Verification....70
Intent Verification....70, 71
Inter-API Communications Layer....163
Interface....17, 65
abstraction....66
access layer....164
behavior....27
derivative....112
model....18, 19, 27
specification....18
system-level....28
timing....141
Interface-Based Design....59
Internal Interrupt....157
Internal Resolution....10
Interrupt....141, 157, 158
external....157
internal....157
Interrupt Request....141
Interrupt Service....141
Interrupt Service Routine....141
Intertask Communication....141
Isolation....132
Jantsch, Axel....5, 33
Kernel Space....141
Kuhn....3
Layout Versus Schematic....91
Leaf Level....20
Library....126
Load....141
Lockout....141
Logic-Level Model....44
Lustre....34
Madisetti....2

Mathematical-Equation Model .36
Mealy Machines....34
Memory
bank-switched....157
cache....158
interface....40
IO....162
mapping....156
protected....156
secondary....156
shared....145, 161
subsystem....159
virtual....156
Memory Management....142
Memory Management Unit....158
Message....67
Message Queues....142
Microcode....16, 51, 154
Microcoded Instruction-Set....142
Mixed-Level Model....31
Model....19, 62, 132
abstract behavioral....39
algorithm....37
architectural....38
basic delay....47
behavioral....19, 26, 82, 92
bus functional....27
cell-level....45
circuit-level....46
classes....23
cycle-callable....43
detailed behavioral....43
detailed performance....47
functional....24
gate-level....45
hardware....42
interface....18, 27
logic-level....44
mathematical-equation....36
mixed-level....31
performance....31, 38
peripheral interconnect....49
power....48
requirements....50
RTL....44
software....49
structural....19, 26
stub....93
switch-level....46
system....35
timing analysis....48
Model Checker....74, 85
Model Interoperability....65
Modeling Concepts....22
Models of Computation....32
Module....55
Monitor....73, 85
Moore Machines....34
Multiprocessor....142
Multiprocessor Architecture....159
Multitasking....142
Multithreading....142
Mutex....142
Netlist....143
Object Code....17, 52
Off-Line Test....143
Open SystemC Initiative....xix, 23
Operating System....156
Operation....67
Operational Test....63
OS Layer....164
Packet....67
Packet Chain....67
Page Fault....143
Path Coverage....81, 85
PBD....103
business and economics....107
components and features....107
development and integration tools....107
support practices....107
Performance Model....31
implementation level....47
token based....31

Peripheral Interconnect Model..49
Petri Nets.....34
Physical Prototype 60, 79
Physical Verification.....91
Pipe 143
PLA.....143, 154
Platform 131
  complexity levels 132
  core 109
  evolution 132
  interface 111
  levels.....131
  object.....132
  object complexity.....108
  provider.....131
  set of blocks 109
  SoC 109
  specification.....113
  taxonomy 108, 131
Platform Based Design ...103, 131
Platform Specification
  application-driven 116
  approach.....125, 132
  architecture-driven 115
  attributes.....116
  bottom-up.....114
  evolution of.....129
  functionality.....125
  market 127
  middle out 115
  structure 126
  technology driven 114
  top down 116
Platform-Based Development
  System.....132
Polling.....143
Port.....66
Portability.....143
Power Model.....48
Precision.....10, 53
Pre-emptive Scheduling.....143
Primitive Function Layer.....163
Priority 143
Product Family.....133
Product Family Approach.....133
Product Family Architecture...133
Product Line.....133
Programmable Logic Code 153
Programmers View 25
Programmers View with Timing
 26
Property.....82
Property Checker 74, 85
Protected Memory 156
Protocol.....66, 144
Protocol Block 66
Protocol Checker 73, 85
Prototype.....60
  physical.....60
  virtual.....61
Pseudo-Code 50
Pseudo-Random Simulation 85
Ptolemy.....35
Queue.....144
Random Pattern 72
Random Simulation 85
RASSP .....xix, 3
Rate Monotonic Analysis.....144
Real-Time 144, 150
Real-Time Clock.....157
Real-Time Operating System..165
Reconfigurable Prototype ...80, 85
Register Shielding.....162
Register Transfer Level 44, 85
Regression Test.....88
Requirement Model 50
Requirement Specification.....64
Resolution 10
  data.....13
  external 10
  functional 14
  internal 10
  software.....16
  structural 15

temporal 12
Retargeting 149
Reusability 65
Reuse 149
RTL 44
RTWG 3
Rugby 34
Run-Time 149
Scheduling Models 144
fair share 144
FIFO based 144
priority based 144
round robin 144
Secondary Memory 156
Semaphores 145
Semi-Formal Verification ... 77, 84
Set of Blocks 109, 110
Shared Memory 145
Shielding layer 145
Signal 34
Signal Coverage 81, 86
Signature Analysis 63, 85
Simple Memory Mapping 156
Simulation 58, 62
cycle 71
cycle-based 83
deterministic 71, 87
directed 83
event-based 84
pseudo random 85
random 85
symbolic 76, 86
SoC Platform 109, 111
Socket 145
Software 57
assembly code 51
microcode 51
object code 52
pseudo code 50
Software Engineering Institute 106, 171
Software Layering 145
Software Model 49
Software Programming Resolution 16
Specification 63
design 64
executable 36, 64
executable design 64
executable requirement 64
requirement 64
Stack 145
Starvation 145
Statement Coverage 81, 86
Static Functional Verification... ..... 74, 86
Structural Model 19, 26
Structural Resolution 15
Stub Model 93
Switch Level Model 46
Symbol Key 22
Symbolic Simulation 76, 86
Synchronous Dataflow 33
Synthesis 58
System 54
SystemC 24
System-Level Interface 28
System-on-Chip 86
Task 146
Task-Switch Latency 146
Taxonomy
definition 1, 9
hardware dependent software .6
model 5
platform based design 6
verification 5
Taxonomy Axes
data value 9
functional 9
structural 9
temporal 9
Technology Foundation .. 128, 131
Technology-Driven 128
Temporal Resolution 12

cycle-accurate 12
cycle-approximate 12
gate propagation 13
instruction accurate 12
partially ordered 12
system event 12
token cycle 12
Test Migration 88
Test Vector 62
Testbench 62, 86
Theorem Prover 75, 86
Thread 146
Timed Functional 26
Timeout 146
Timer 146
Timing Analysis Model 48
Toggle Coverage 81, 86
Token-Based Performance Model 38
Top-Down Design 59
Transaction 67
Transaction Level Model 24
Triggering Coverage 81, 86
Untimed Functional 25
User Space 147
Variable Bit-Rate 147
Variable Features 133
Verification
classification 69
equivalence 70
integration 70, 94
intent 69
metrics 81, 86
VC 70, 93
View 132
Viewpoint 132
Virtual Component 5, 29, 131
interface 29, 66
Virtual Memory 156
Virtual Prototype 61, 80, 86
Visited State Coverage 81, 86
VSI Alliance 4
VSIA ..... xix, 4
Watchdog Timer 147
Y-chart 2